TMS320C27xx Optimizing C Compiler User's Guide

Preliminary

Literature Number: SPRU212A
March 1998

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 office.

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 *TMS320C27xx Optimizing C Compiler User's Guide* explains how to use these compiler tools:

- ❑ Code generator
- ❑ Library-build utility
- ❑ Optimizer
- ❑ Parser
- ❑ Interlist utility

The TMS320C27xx C compiler accepts American National Standards Institute (ANSI) standard C source code and produces assembly language source code for the TMS320C27xx devices. This user's guide discusses the characteristics of the TMS320C27xx 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.

Notational Conventions

This document uses the following conventions:

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

  Here is a sample of C code:

```
#include <stdio.h>

main()

{
   printf("hello, world\n");
}
```

- In syntax descriptions, the instruction, command, or directive is in a **bold** typeface and parameters are in *italics*. Portions of a syntax that are in bold must be entered as shown; portions of a syntax that are in italics describe the type of information that should be entered. Syntax that is entered on a command line is centered in a bounded box:

  cl27 [*options*] [*filenames*] [*–z* [*link_options*] [*object files*]]

  Syntax used in a text file is left justified in a bounded box:

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

- 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. This is an example of a command that has an optional parameter:

  ac27*inputfile* [*outputfile*] [*options]*

  The ac27 command has three parameters. The first parameter, *inputfile*, is required. The second and third parameters, *outputfile* and *options*, are optional.

- Braces ({ and }) indicate that you must choose one of the parameters within the braces; you don't enter the braces themselves. This is an example of a command with braces; the braces are not included in the actual syntax but indicate that you must specify either the –c or –cr option:

  lnk27 {**–c** | **–cr**} *filenames* [**–o** *name.out*] **–l** *libraryname*

- The TMS320C27xx core is referred to as TMS320C27xx and 'C27xx.

Related Documentation From Texas Instruments

The following books describe the TMS320C27xx and related support tools. To obtain any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, identify the book by its title and literature number (located on the title page):

TMS320C27xx DSP CPU and Instruction Set Reference Guide (literature number SPRU220) describes the central processing unit (CPU) and the assembly language instructions of the TMS320C27xx 16-bit fixed-point digital signal processors (DSPs). It also describes emulation features available on these DSPs.

TMS320C27xx Code Generation Tools Getting Started Guide (literature number SPRU213) describes how to install the TMS320C27xx assembly language tools and the C compiler for the TMS320C27xx device. The installation for MS-DOS™, SunOS™, and HP-UX™ 9.0x systems are covered.

TMS320C27xx Assembly Language Tools User's Guide (literature number SPRU211) describes the assembly language tools (assembler and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the TMS320C27xx device.

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

TMS320C27xx Simulator Getting Started (literature number SPRU216) describes how to install the simulator and the C source debugger for the TMS320C27xx device. The installation for MS-DOS™, SunOS™, and HP-UX™ systems are covered.

TMS320C27xx Emulator Getting Started (literature number SPRU215) describes how to install the emulator software and the C source debugger for the TMS320C27xx device. The installation for MS-DOS™, SunOS™, and HP-UX™ systems are covered.

TMS320C27xx C Source Debugger User's Guide (literature number SPRU214) tells you how to invoke the TMS320C27xx emulator 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. It also includes a tutorial that introduces basic debugger functionality.

Related Documentation

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

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

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

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

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.

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

Trademarks

Intel, i286, i386, and i486 is a trademark of Intel Corporation.

MCS-86 is a trademark of Intel Corporation.

Motorola-S is a trademark of Motorola, Inc.

Motorola is a trademark of Motorola, Inc.

Tektronix is a trademark of Tektronix, Inc.

Windows and Windows NT are registered trademarks of Microsoft Corporation.

If You Need Assistance . . .

❑ World-Wide Web Sites

TI Online	http://www.ti.com
Semiconductor Product Information Center (PIC)	http://www.ti.com/sc/docs/pic/home.htm
DSP Solutions	http://www.ti.com/dsps
320 Hotline On-line™	http://www.ti.com/sc/docs/dsps/support.htm

❑ North America, South America, Central America

Product Information Center (PIC)	(972) 644-5580		
TI Literature Response Center U.S.A.	(800) 477-8924		
Software Registration/Upgrades	(214) 638-0333	Fax: (214) 638-7742	
U.S.A. Factory Repair/Hardware Upgrades	(281) 274-2285		
U.S. Technical Training Organization	(972) 644-5580		
DSP Hotline	(281) 274-2320	Fax: (281) 274-2324	Email: dsph@ti.com
DSP Modem BBS	(281) 274-2323		

DSP Internet BBS via anonymous ftp to ftp://ftp.ti.com/pub/tms320bbs

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32

Email: epic@ti.com

Deutsch	+49 8161 80 33 11 or +33 1 30 70 11 68
English	+33 1 30 70 11 65
Francais	+33 1 30 70 11 64
Italiano	+33 1 30 70 11 67

EPIC Modem BBS	+33 1 30 70 11 99	
European Factory Repair	+33 4 93 22 25 40	
Europe Customer Training Helpline		Fax: +49 81 61 80 40 10

❑ Asia-Pacific

Literature Response Center	+852 2 956 7288	Fax: +852 2 956 2200
Hong Kong DSP Hotline	+852 2 956 7268	Fax: +852 2 956 1002
Korea DSP Hotline	+82 2 551 2804	Fax: +82 2 551 2828
Korea DSP Modem BBS	+82 2 551 2914	
Singapore DSP Hotline		Fax: +65 390 7179
Taiwan DSP Hotline	+886 2 377 1450	Fax: +886 2 377 2718
Taiwan DSP Modem BBS	+886 2 376 2592	

Taiwan DSP Internet BBS via anonymous ftp to ftp://dsp.ee.tit.edu.tw/pub/TI/

❑ Japan

Product Information Center	+0120-81-0026 (in Japan)	Fax: +0120-81-0036 (in Japan)
	+03-3457-0972 or (INTL) 813-3457-0972	Fax: +03-3457-1259 or (INTL) 813-3457-1259
DSP Hotline	+03-3769-8735 or (INTL) 813-3769-8735	Fax: +03-3457-7071 or (INTL) 813-3457-7071
DSP BBS via Nifty-Serve	Type “Go TIASP”	

❑ Documentation

When making suggestions or reporting errors in documentation, please include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail: Texas Instruments Incorporated
Technical Documentation Services, MS 702
P.O. Box 1443
Houston, Texas 77251-1443

Email: dsph@ti.com

Note: When calling a Literature Response Center to order documentation, please specify the literature number of the book.

Contents

Figures

Tables

Examples

Chapter 1

Introduction

The TMS320C27xx is fully supported by a complete set of code generation tools, including an optimizing C compiler, assembler, linker, archiver, and software simulator.

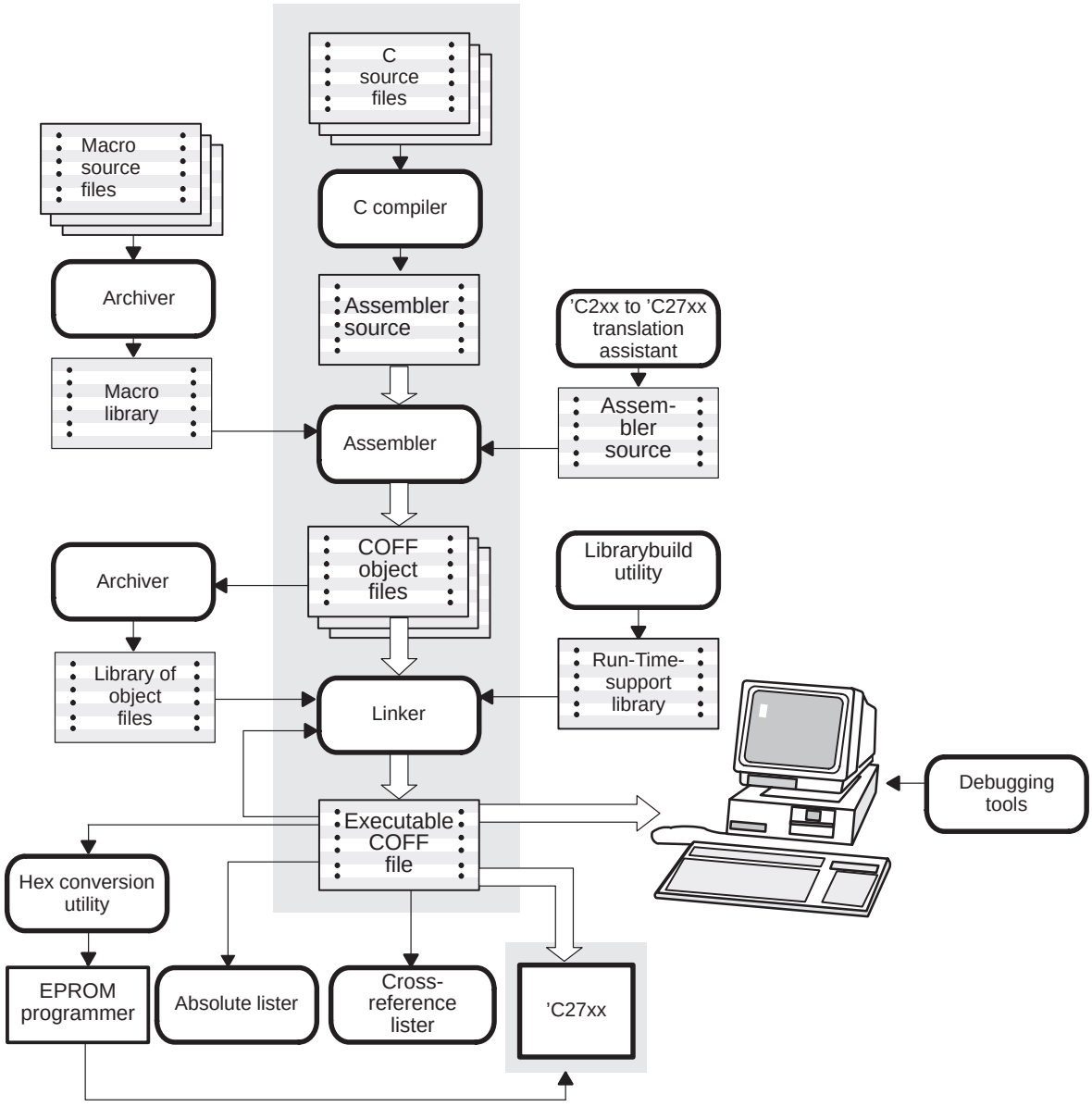
This chapter provides an overview of these tools and introduces the features of the optimizing C compiler.

Topic **Page**

1.1 Software Development Tools Overview

Figure 1–1 illustrates the ’C27xx software development flow. The shaded portion of the figure highlights the most common path of software development for C language programs. The other portions are peripheral functions that enhance the development process.

Figure 1–1. TMS320C27xx Software Development Flow



1.2 Tools Descriptions

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

- The **C compiler** translates C source code into ’C27xx assembly language source code. A **shell program**, an **optimizer**, and an **interlist utility** are included in the compiler package.
  - The shell program enables you to compile, assemble, and link source modules.
  - The optimizer modifies code to improve the efficiency of C programs.
  - The interlist utility incorporates C source statements with assembly language output.

  See Chapter 2, *Using the C Compiler*, for information about how to invoke the C compiler, the shell, the optimizer, and the interlist utility.

- The **assembler** translates assembly language source files into machine language object files. The machine language is based on common object file format (COFF). The *TMS320C27xx Assembly Language Tools User’s Guide* explains how to use the assembler.

- The **linker** combines object files into a single executable object module. As it creates the executable module, it adjusts references to symbols and resolves external references. The linker accepts relocatable COFF object files and object libraries as input. The *TMS320C27xx Assembly Language Tools User’s Guide* explains the linker in detail.

- The **archiver** collects a group of files, source or object, 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 (object library). An object library containing the compiled RTS functions is shipped with the C compiler. The *TMS320C27xx Assembly Language Tools Guide* explains how to use the archiver.

- The **library-build utility** builds your own customized, run-time-support library (see Chapter 8, *Library-Build Utility*). Standard run-time-support library functions are provided as source code in rts.src and as object code in rts.lib.

  The run-time-support library contains the ANSI standard run-time-support functions, compiler-utility functions, floating-point arithmetic functions, and C I/O functions that are supported by the ’C27xx compiler.

- The ’C27xx debugger accepts executable COFF files as input, but most EPROM programmers do not. The **hex conversion utility** converts a COFF object file into TI-Tagged, ASCII-hex, Intel, Motorola-S, or Tektronix object format. You can download the converted file to an EPROM programmer. The *TMS320C27xx Assembly Language Tools User’s Guide* explains how to use the hex conversion utility.

- The **absolute lister** uses linked object files to produce an assembly listing that provides final addresses for symbol references and code. Using a linked object file as input, the absolute lister produces an intermediate .abs file that the assembler can use. For more information about the absolute lister, see the *TMS320C27xx Assembly Language Tools User’s Guide*.

- The **translation assistant** accepts ’C2xx assembly source input and produces a ’C27xx assembly source file. Instructions that are not directly translatable are flagged with warning messages to help you complete the translation.

- The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definitions, and their references across all linked source files. The *TMS320C27xx Assembly Language Tools User’s Guide* explains how to use the cross-reference utility.

- The **post-link optimizer** removes or modifies assembly language instructions to generate better code. The post-link optimizer must be run with the shell option –plink. For more information, see Chapter 8 of the *TMS320C27xxAssembly Language Tools User’s Guide*.

The purpose of this development process is to produce a module that can be executed in a ’C27xx target system. You can use a simulator or emulator to refine and correct your code.

For information about the debugging tools, see the *TMS320C27xx C Source Debugger User’s Guide*.

1.3 C Compiler Overview

The ’C27xx C compiler is a full-featured optimizing compiler that translates standard ANSI C programs into ’C27xx assembly language source. The following list describes key characteristics of the compiler.

- **ANSI standard C**

  The ’C27xx compiler fully conforms to the ANSI C standard as defined by ANSI and described in Kernighan and Ritchie’s *The C Programming Language* (second edition).

- **ANSI standard runtime support**

  The compiler tools come with a complete runtime library. The library includes functions for standard input and output, string manipulation, dynamic memory allocation, data conversion, timekeeping, and trigonometry, plus exponential and hyperbolic functions. Functions for signal handling are not included, because these are target-system specific. for more information see Chapter 7, *Run-Time-Support Functions*.

- **Assembly source output**

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

- **COFF object files**

  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 support for source-level debugging.

- **Code to initialize data into ROM**

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

- **Compiler shell program**

  The compiler package includes a shell program that you use to compile, assemble, and link programs in a single step. For more information, see section 2.1, *About the Shell Program*, on page 2-2.

- **Flexible assembly language interface**

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

- **Integrated preprocessor**

  The C preprocessor is integrated with the parser, which decreases compilation time. Stand-alone preprocessing or preprocessed listing is also available. For more information, see section 2.3, *Controlling the Preprocessor*, on page 2-19.

- **Optimization**

  The compiler uses a sophisticated optimization pass that employs several advanced techniques for generating efficient, compact code from C source. The compiler performs general and 'C27xx-specific optimizations. General optimizations can be applied to any C code. 'C27xx-specific optimizations take advantage of the features unique to the 'C27xx architecture. For detailed information about the C compiler's optimization techniques, see section 3.1, *Using the C Compiler Optimizer*, on page 3-2.

- **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, see section 2.5, *Using the Interlist Utility*, on page 2-28.

- **Library-build utility**

  The compiler package has a utility which can create object libraries from archived source libraries in one step. This is useful for recompiling the RTS library using compiler options that fit your needs. The complete RTS library source is included with the compiler product.

Chapter 2

Using the C Compiler

Translating your source program into code that the ’C27xx can execute is a multistep process. You must compile, assemble, and link your source files to create an executable object file. The ’C27xx compiler tools contain 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 shell program to compile, assemble, and link your programs.

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

- During compilation, your code is run through the preprocessor, which is part of the parser. You can control the preprocessor with macros and various other preprocessor directives.
- The C compiler includes an optimizer that can be optionally invoked to allow you to produce highly optimized code.
- Inline function expansion allows you to save the overhead of a function call and enable further optimizations.
- The compiler tools include 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.

Topic **Page**

2.1 About the Shell Program

The compiler shell program (cl27) lets you compile, assemble, and optionally link in one step. The shell runs one or more source modules through the following tools:

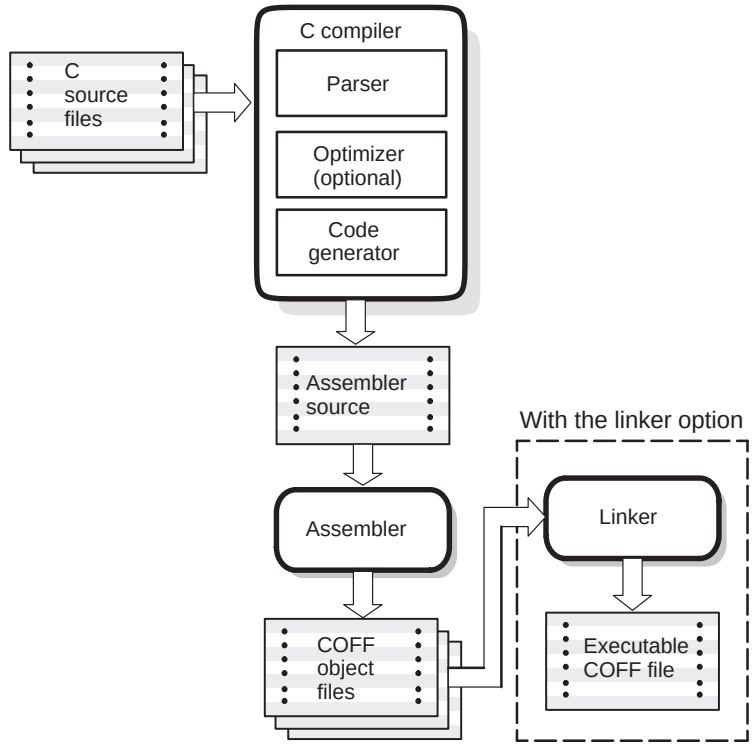
1) The **compiler**, which includes the parser, optimizer, and code generator, accepts C source code and produces 'C27xx assembly language source code.

2) The **assembler** generates a COFF object file.

3) The **linker** links your files to create an executable object file. The linker is optional at this point. You can compile and assemble various files with the shell and link them later. See Chapter 4, *Linking C Code*, for information about linking the files in a separate step.

By default, the shell compiles and assembles files; however, you can also link the files using the –z shell option. Figure 2–1 illustrates the path the shell takes with and without using the linker.

Figure 2–1. The Shell Program Overview



For complete descriptions of the assembler and linker, see the *TMS320C27xx Assembly Language Tools User's Guide*.

2.1.1 Invoking the C Compiler Shell

To invoke the compiler shell, enter:

cl27 [*–options*] *filenames* [*object files*] [**–z** [*link_options*]]

cl27	Command that runs the compiler and the assembler
options	Options that affect the way the shell processes input files (the options are listed in Table 2–1 on page 2-7).
filenames	One or more C source files, assembly language source files, linear assembly files, or object files
object files	Name of the additional object files for the linking process
–z	Option that invokes the linker. See Chapter 4, *Linking C Code*, for more information about invoking the linker.
link_options	Options that control the linking process

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

```
cl27 –q symtab file seek.asm
```

As cl27 encounters each source file, it prints the C filenames in square brackets ([ ]), assembly language filenames in angle brackets (< >). This example uses the –q option to suppress the additional progress information that cl27 produces. Entering the command above produces these messages:

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

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

```
% cl27 symtab
[symtab]
TMS320C27xx ANSI C Compiler      Version xx
Copyright (c) 1997 Texas Instruments Incorporated
   ”symtab.c”      ==>     symtab
TMS320C27xx ANSI C Codegen    Version xx
Copyright (c) 1997 Texas Instruments Incorporated
   ”symtab.c”:     ==>     symtab
TMS320C27xx COFF Assembler    Version xx
Copyright (c) 1997 Texas Instruments Incorporated
 PASS 1
 PASS 2

No Errors, No Warnings
```

2.2 Changing the Compiler's Behavior With Options

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

The following apply to the compiler options:

- Options are either single letters or 2-letter pairs.
- Options are *not* case sensitive.
- Options are preceded by a hyphen.
- Single-letter options without parameters can be combined. For example, –sgq is equivalent to –s –g –q.
- Two-letter pair options that have the same first letter can be combined. For example, –pe and –pk can be combined as –pek.
- Options that have parameters, such as –u*name* and –i*directory*, cannot be combined. They must be specified separately.
- Options with parameters can have a space between the option and the parameter, or they can be right next to each other with no intervening space.
- Files and options except for the –z option can occur in any order. The –z option must follow all other compiler options and precede any linker options.

You can define default options for the shell by using the C_OPTION environment variable.

Table 2–1 summarizes all options (including linker options). Use the page numbers in the tables to locate more complete descriptions of the options.

For an online summary of the options, enter **cl27** with no parameters on the command line.

Table 2–1. Shell Options Summary

(a) Options that control the compiler shell

Option	Effect	Page
–@*filename*	Interprets the contents of a file as an extension to the command line	2-11
–b	Generates a user information file	2-11
–c	Disables linking (negates –z)	2-11, 4-3
–d*name*[=*def*]	Predefines *name*	2-11
–g	Enables symbolic debugging	2-11
–i*directory*	Defines #include search path	2-12, 2-20
–k	Keeps the assembly language (.asm) file	2-12
–n	Compiles or assembly optimizes only	2-13
–plink	Modifies assembly instructions after using the –z option	4-4†
–q	Suppresses progress messages (quiet)	2-13
–qq	Suppresses all messages (super quiet)	2-13
–s	Interlists optimizer comments (if available) and assembly source statements; otherwise interlists C and assembly source statements	2-13
–ss	Interlists optimizer comments with C source and assembly statements	2-13, 3-13
–u*name*	Undefines *name*	2-14
–z	Enables linking	2-14

† See the *TMS320C27xx Assembly Language User's Guide* for more information.

(b) Options that overlook ANSI C type-checking

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

Table 2–1. Shell Options Summary (Continued)

(c) Options that are machine-specific

Option	Effect	Page
–ma	Assumes aliased variables	2-12
–mf	Optimizes for code speed over size	2-12
–mn	Enables optimizer options disabled by –g	2-12
–mt	Generates code to access switch tables from data space	2-13

(d) Options that control the parser

Option	Effect	Page
–pg	Enables trigraph expansion	2-22
–pe	Treats code-E errors as warnings	2-30
–pf	Generates function prototype listing file	2-22
–pk	Allows K&R compatibility	5-16
–pl	Generates preprocessed listing (.pp file)	2-21
–pm	Combines source files to perform program-level optimization	3-6
–pn	Suppresses #line directives in .pp file	2-21
–po	Preprocesses only	2-21
–pr	Generates an error listing	2-30
–pw0	Disables all warning messages	2-30
–pw1	Enables serious warning messages (default)	2-30
–pw2	Enables all warning messages	2-30
–px*filename*	Names the output file created when using the –pm option	3–9

Table 2–1. Shell Options Summary (Continued)

(e) Options that control optimization

Option	Effect	Page
–o0	Optimizes registers	3-2
–o1	Uses –o0 optimizations *and* optimizes locally	3-3
–o2 or –o	Uses –o1 optimizations *and* optimizes globally	3-3
–o3	Uses –o2 optimizations *and* optimizes the file	3-3
–oi*size*	Sets automatic inlining size (–o3 only)	3-12
–ol0	Informs the optimizer that your file alters a standard library function	3-4
–ol1	Informs the optimizer that your file declares a standard library function	3-4
–ol2	Informs the optimizer that your file does not declare or alter library functions. Overrides the –ol0 and –ol1 options	3-4
–on0	Disables optimizer information file	3-5
–on1	Produces optimizer information file	3-5
–on2	Produces verbose optimizer information file	3-5
–op0	Specifies that the module contains functions and variables that are called or modified from outside the source code provided to the compiler	3-6
–op1	Specifies that the module contains variables modified from outside the source code provided to the compiler but does not use functions called from outside the source code	3-6
–op2	Specifies that the module contains no functions or variables that are called or modified from outside the source code provided to the compiler (default)	3-6
–op3	Specifies that the module contains functions that are called from outside the source code provided to the compiler but does not use variables modified from outside the source code	3-6
–os	Interlists optimizer comments with assembly statements	3-13

Table 2–1. Shell Options Summary (Continued)

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

Option	Effect	Page
–x0	Disables inline expansion of intrinsic operators	2-24
–x1	Enables inline expansion of intrinsic operators (default)	2-24
–x2 or –x	Defines the symbol _INLINE and invokes the optimizer with –o2	2-24

(g) Options that control the assembler

Option	Effect	Page
–aa	Enables absolute listing	2-18
–ad *symbol*	Defines the symbol	2-18
–ahc *fileneame*	Copies filenames	2-18
–ahi *fileneame*	Includes filenames	2-18
–al	Generates an assembly listing file	2-18
–as	Puts labels in the symbol table	2-18
–ax	Generates the cross-reference file	2-18

(h) Options that change the default file extensions

Option	Effect	Page
–ea[.]*extension*	Sets default extension for assembly source files	2-15
–eo[.]*extension*	Sets default extension for object files	2-15

(i) Options that specify files

Option	Effect	Page
–fa*filename*	Changes how assembler source files are identified	2-15
–fc*filename*	Changes how C source files are identified	2-15
–fo*filename*	Changes how object code is identified	2-15

Table 2–1. Shell Options Summary (Continued)

(j) Options that specify directories

Option	Effect	Page
–fr*directory*	Specifies object file directory	2-16
–fs*directory*	Specifies assembly file directory	2-16
–ft*directory*	Specifies temporary file directory	2-16

2.2.1 Frequently Used Options

Following are detailed descriptions of options that you will probably use frequently:

–@*filename* Uses the contents of the specified file instead of command line entries. You can use this command to avoid limitations on command line length imposed by the host operating system. Use a # or ; in the command file to include comments.

–b Generates an auxiliary information file that you can refer to for information about stack size and function calls. The filename is the C source filename with a .aux extension.

–c Suppresses the linker and overrides the –z option which specifies linking. Use this option when you have –z specified in the C_OPTION environment variable and you don't want to link. For more information, see section 4.2.1, *Disabling the Linker (–c Shell Option)*, on page 4-3.

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

–g Generates symbolic debugging directives that are used by the C source-level debugger and enables assembly source debugging in the assembler. The –g option disables many code generator optimizations, because they disrupt the debugger. You can use the –g option with the –o option to maximize the amount of optimization that is compatible with debugging (see section 3.7, *Debugging Optimized Code*, on page 3-16.

–i*directory* Adds *directory* to the list of directories that the compiler searches for #include files. You can use this option a maximum of 64 times to define several directories; be sure to separate –i options with spaces. If you do not specify a directory name, the preprocessor ignores the –i option. For more information, see section 2.3.2.1, *Changing the #include File Search Path(–i Option)*, on page 2-20.

–k Retains the assembly language output from the compiler. Normally, the shell deletes the output assembly language file after assembly is complete.

–ma Assumes that variables are aliased. The compiler assumes that pointers may alias (point to) named variables. Therefore, it disables register optimizations when an assignment is made through a pointer if the compiler determines that there may be another pointer pointing to the same object.

–mf Optimizes your code for speed over size. By default, the 'C27xx optimizer attempts to reduce the size of your code at the expense of speed.

–mn Reenables the optimizations disabled by the –g option. If you use the –g option, many code generator optimizations are disabled because they disrupt the debugger. Therefore, if you use the –mn option, portions of the debugger's functionality will be unreliable.

–mt Generates code to access switch tables from data space instead of from program space. This reduces code needed to access the switch tables. Use this option if your code consists of switch statements consisting of a large number of cases (more than about 20).

For large switch statements the code generator produces a switch table that enables efficient branches to the appropriate case label. This table is placed in the compiler-created section called .switch. By default, this section is linked into program space. The code generator then uses PREAD instructions to access the switch table.

You can optionally link the .switch section into data space to save program memory or to take advantage of memory that can be configured as program and data. In this case, use the –mt switch to inform the code generator *not* to use PREAD instructions to access the switch table and to use more efficient data memory instructions instead.

If your C program does not contain switch constructs with more than about 20 case labels, then this issue is not a concern.

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

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

–qq Suppresses all output except error messages.

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

–ss Invokes the interlist utility, which interweaves original C source with compiler-generated assembly language. This option might reorganize your code substantially. For more information, see section 2.5, *Using the Interlist Utility,* on page 2-28.

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

–z Runs the linker on the specified object files. The –z option and associated linker command options follow all other options on the command line. All arguments that follow –z are passed to the linker. For more information, see Chapter 4, *Linking C Code*.

2.2.2 Specifying Filenames

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

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

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

For information about how you can alter the way that the shell interprets individual filenames, see section 2.2.3, *Changing How the Shell Program Interprets Filenames (–fa, –fc, and –fo Options*). For information about how you can alter the way that the shell interprets and names the extensions of assembly source and object files, see section 2.2.5, *Specifying Directories*, on page 2-16.

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

```
cl27 *.c
```

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

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

–fa*filename*	for an assembly language source file
–fc*filename*	for a C source file
–fo*filename*	for an object file

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

```
cl27 –fc file.s –fa assy
```

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

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

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

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

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

```
cl27 –ea .rrr –eo .o fit.rrr
```

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

```
cl27 –earrr –eoo fit.rrr
```

2.2.5 Specifying Directories

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

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

```
cl27 –fr d:\object
```

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

```
cl27 –fs d:\assembly
```

–ft Specifies a directory for temporary intermediate files. The –ft option overrides the TMP environment variable. To specify a temporary directory, type the directory's pathname on the command line after the –ft option:

```
cl27 –ft c:\temp
```

2.2.6 Options That Overlook ANSI C Type-Checking

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

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

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

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

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

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

- Pointers to differently qualified types can be combined:

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

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

2.2.7 Options That Control the Assembler

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

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

–ad *symbol* Tells the assembler to define the specified symbol for the assembly module.

–ahc *filename* Copies filename before any source statements from the input file are assembled. The assembler treats the–ahc option file in a manner similar to .copy files; the statements that are assembled from the copied file are not printed in the assembly listing.

–ahi *filename* Includes filename before any source statements from the input file are assembled. The assembler treats the –ahi option file in a manner similar to .include files; the statements that are assembled from the included file are not printed in the assembly listing.

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

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

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

For more information about assembler options, see the *TMS320C27xx Assembly Language Tools User's Guide.*

2.3 Controlling the Preprocessor

This section describes specific features that control the ’C27xx preprocessor, which is part of the parser. A general description of C preprocessing is presented in section A12 of K&R. The ’C27xx C compiler includes standard C preprocessing functions, which are built into the first pass of the compiler. The preprocessor handles:

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

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

2.3.1 Predefined Macro Names

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

Table 2–2. Predefined Macro Names

Macro Name	Description
__LINE__†	Expands to the current line number
__FILE__†	Expands to the current source filename
__DATE__†	Expands to the compilation date in the form *mm dd yyyy*
__TIME__†	Expands to the compilation time in the form *hh:mm:ss*
_INLINE	Expands to 1 under the –x or –x2 option; undefined otherwise
_TMS320C27xx	Expands to 1 (identifies the ’C27xx processor)

† Specified by the ANSI standard

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

```
printf ( ”%s %s” , __TIME__ , __DATE__);
```

translates to a line such as:

```
printf (”%s %s” , ”13:58:17”, ”Jan 14 1997”);
```

2.3.2 The Search Path for #include Files

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

- ❏ If you enclose the filename in double quotes (“ ”), the compiler searches for the file in the following directories in the order in which they are listed:
  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 option
  3) Directories set with the C_DIR environment variable
- ❏ If you enclose the filename in angle brackets (< >), the compiler searches for the file in the following directories in the order in the order in which they are listed:
  1) Directories named with the –i option
  2) Directories set with the C_DIR environment variable

See section 2.3.2.1, *Changing the #include File Search Path (–i Option*) for information on using the –i option.

2.3.2.1 Changing the #include File Search Path (–i Option)

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

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

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

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

Assume that the complete pathname for alt.h is:

Windows	c:\320tools\files\alt.h
UNIX	/320tools/files/alt.h

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

Operating System	Enter
Windows	`cl27 –ic:\320tools\files source.c`
UNIX	`cl27 –i/320tools/files source.c`

2.3.3 Generating a Preprocessed Listing File (–pl Option)

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

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

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

The preprocessed output file contains no preprocessor directives other than #line; the compiler inserts #line directives to synchronize line and file information in the output files with input position from the original source files. You can use the –pn option to suppress #line directives (see section 2.3.3.2, *Removing the #line Directives From the Preprocessed Listing File*).

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

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

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

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

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

2.3.4 Creating Custom Error Messages With the #warn and #error Directives

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

#error *token-sequence*

#warn *token-sequence*

2.3.5 Enabling Trigraph Expansion (–pg Option)

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

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

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

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

2.4 Using Inline Function Expansion

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

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

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

- ❑ Intrinsic operators are expanded by default.
- ❑ Automatic inline function expansion is performed on small functions that are invoked by the compiler with the –o3 option. For information about automatic inline function expansion, see section 3.5.
- ❑ Definition-controlled inline expansion is performed when you invoke the compiler with optimization (–x option) and the compiler encounters the inline keyword in code.

Note: Function Inlining Can Greatly Increase Code Size

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

2.4.1 Inlining Intrinsic Operators

There are many intrinsic operators for the 'C27xx. All of them are automatically inlined by the compiler. The inlining occurs whether or not you use the optimizer and whether or not you use any compiler or optimizer options on the command line.

You can stop the inlining by invoking the compiler with the –x0 option.

For more information about intrinsics, see section 6.4.5, *Using Intrinsics to Access Assembly Language Statements*, on page 6-22.

2.4.2 Controlling Inline Function Expansion (–x Option)

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

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

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

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

2.4.3 Using Definition-Controlled Inline Function Expansion

Definition-controlled inline expansion is performed when you invoke the compiler with optimization and the compiler encounters the inline keyword in code. Functions with local static variables or a variable number of arguments are not inlined, with the exception of functions declared as static inline. In functions declared as static inline, the 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 in few places (though the compiler does not enforce this). You can control this type of function inlining with the inline keyword.

The inline keyword specifies that a function is expanded inline at the point at which it is called rather than by using standard calling procedures. You can use the inline keyword in two ways:

- By declaring a function as inline within a module
- By declaring a function as static inline

2.4.3.1 Declaring a Function as Inline Within a Module

By declaring a function as inline within a module (with the inline keyword), you can specify that the function is inlined within that module. A global symbol for the function is created, but the function is inlined only within the module in which it is declared as inline. It is not called by other modules unless the modules contain a compatible static inline declaration.

Functions declared as inline are expanded when the optimizer is invoked and the –x0 option is used. Using the –x2 option automatically invokes the optimizer at the default level (–o2).

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

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

2.4.3.2 Declaring a Function as Static Inline

Declaring a function as static inline in a header file specifies that the function is inlined in any module that includes the header. The declaration names the function and specifies that the function 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.

Use this syntax to declare a function as static inline:

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

2.4.4 The _INLINE Preprocessor Symbol

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

Example 2–1 on page 2-26 illustrates how the run-time-support library uses the _INLINE preprocessor symbol. The _INLINE preprocessor symbol is used in the string.h header file to declare the function correctly, regardless of whether inlining is used. The _INLINE preprocessor symbol conditionally defines _ _INLINE so that strlen is declared as static inline only if the _INLINE preprocessor symbol is defined.

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

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

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

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

(a) string.h

```
#ifndef _INLINE
#define __INLINE static inline
#else
#define__INLINE
#endif

__INLINE size_t  strlen(const char *s1);

#ifdef _INLINE

static inline size_t strlen(const char *string
{
       register const char *rstr = string;
        register       size_t n  = 0;

        while (*rstr++) ++n;
         return n;
}

#endif /*_INLINE*/
#undef __INLINE
```

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

(b) strlen.c

```
/*************************************************************************/
/*  strlen.c                                                             */
/*************************************************************************/
#undef _INLINE                                  /*DISABLE INLINE EXPANSION*/
#include <string.h>

size_t strlen(const char *string)
{
   register const char *rstr = string;
    register      size_t n   = 0;

   while (*rstr++) ++n;
   return (n);
}
```

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

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

2.5 Using the Interlist Utility

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

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

```
cl27 –s function
```

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

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

Example 2–2 shows a typical interlisted assembly file.

Example 2–2. An Interlisted Assembly Language File

```
_main:
;----------------------------------------------------------------------
;   5 | printf (”Hello, world\n”);
;----------------------------------------------------------------------
        MOV       AR3,#SL1
        LC        #_printf
        ; call occurs [#_printf]
;----------------------------------------------------------------------
;   6 | return 0;
;----------------------------------------------------------------------
        MOV       AL,#0
;----------------------------------------------------------------------
;   7 | }
;----------------------------------------------------------------------
        LRET
        ; branch occurs
;*************************************************************
;* STRINGS                                                   *
;*************************************************************
   .sect  ”.const”
SL1:   .string   ”Hello, world”,10,0
;*************************************************************
;* UNDEFINED EXTERNAL REFERENCES                             *
;*************************************************************
   .global   _printf
```

For more information about using the interlist utility with the optimizer, see section 3.6, *Using the Interlist Utility With the Optimizer*, on page 3-13.

2.6 Understanding and Handling Compiler Errors

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

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

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

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

- **Code-W errors** are warnings resulting from a condition that is technically undefined according to the rules of the language or that can cause unexpected results. The compiler continues running.
- **Code-E errors** are recoverable, resulting from a condition that violates the semantic rules of the language. Although these are normally fatal errors, the compiler can recover and generate an output file if you use the –pe option. See section 2.6.2, *Treating Code-E Errors as Warnings (–pe Option)*, on page 2-30, for more information.
- **Code-F errors** are fatal, resulting from a condition that violates the syntactic or semantic rules of the language. The compiler cannot recover and does not generate output for code-F errors.
- **Code-I errors** are implementation errors, occurring when one of the compiler's internal limits is exceeded. These errors are usually caused by extreme behavior in the source code rather than by explicit errors. In most cases, code-I errors cause the compiler to abort immediately. Most code-I messages contain the maximum value for the limit that was exceeded. (Those limits that are absolute are listed in section 5.9, *Compiler Limits*, on page 5-18.)
- **Other error messages**, such as incorrect command-line syntax or inability to find specified files, are usually fatal. They are identified by the symbol >> preceding the message.

Table 2–3. Example Error Messages

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

2.6.1 Generating an Error Listing (–pr Option)

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

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

A *fatal error* prevents the compiler from generating an output file. Normally, code-E, -F, and -I errors are fatal, while -W errors are not. The –pe option causes the compiler to treat code-E errors as warnings, so that the compiler generates code for the file despite the error. There is no way to specify recovery from code-F or -I errors. These are always fatal.

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

See section 2.6.4, *An Example of How You Can Use Error Options*, on page 2-31, for an example of the –pe option.

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

You can determine which levels of warning messages to display by setting the warning message level with the –pw option. The number following –pw denotes the level (0, 1, or 2). Use Table 2–4 to select the appropriate level. See section 2.6.4, *An Example of How You Can Use Error Options*, on page 2-31 for an example of the –pw option.

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

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

2.6.4 How You Can Use Error Options

The following examples demonstrate how you can suppress errors with the –pe option and alter the level of error messages with the –pw option. The examples use this code segment, which is contained in file err.c:

```
   int *pi; char *pc;
func() {
#if STDC
   pi = (int *) pc;
#else
   pi = pc;
#endif
}
```

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

  ```
  [err.c]
  "err.c", line 8: [E127] operands of '=' point to different types
  ```

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

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

  ```
  [err.c]
  "err.c", line8: [E122] operands of '=' point to different types
  ```

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

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

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

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

Chapter 3

Optimizing Your Code

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

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

Topic **Page**

3.1 Using the C Compiler Optimizer

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 cl27 shell program, specifying the –o option on the cl27 command line. (You may also invoke the optimizer outside cl27; see section A.4, *Invoking the Optimizer Individually*, on page A-7 for more information).

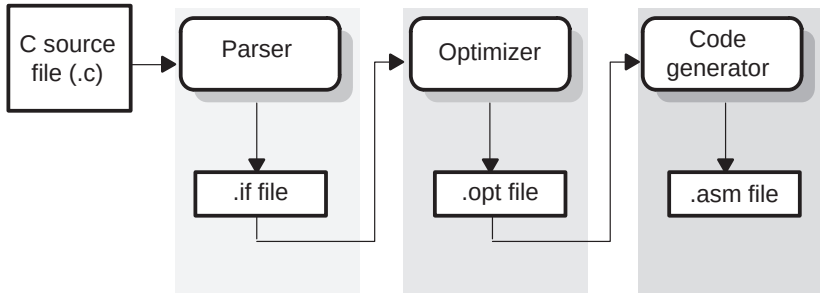
The –o option can be followed by a digit specifying the level of optimization. If you do not specify a level digit, the optimizer defaults to level 2. The –o option can also be followed by the –ol or –on options with modifiers. For more information, see Table 2–1 (d) *Options that control the optimizer*, on page 2-9.

For example, to invoke the compiler using full optimization with inline function expansion, enter:

```
cl27 –o –x2 function.c
```

Figure 3–1 illustrates the execution flow of the compiler with stand-alone optimization.

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



To invoke the optimizer, use the cl27 shell program, specifying the –o option on the cl27 command line with the appropriate digit for the level of optimization (0, 1, 2, and 3), which controls the type and degree of optimization:

❑ **–o0**

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

❑ **–o1**

Performs all –o0 optimizations, and:

- Performs local copy/constant propagation
- Removes unused assignments
- Eliminates local common expressions

❑ **–o2**

Performs all –o1 optimizations, and:

- Performs loop optimizations
- Eliminates global common subexpressions
- Eliminates global unused assignments

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

❑ **–o3**

Performs all –o2 optimizations, and:

- Removes all functions that are never called
- Simplifies functions with return values that are never used
- Inlines calls to small functions
- Identifies file-level variable characteristics

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

These levels of optimization are performed by the stand-alone optimization pass. The code generator performs several additional optimizations, particularly 'C27xx-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.

3.2 Using the –o3 Option

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3.2 Optimization Considerations When Mixing C and Assembly

If you have any assembly functions in your program, exercise caution when using the –pm option. The compiler recognizes only the C source code and not any assembly code that might be present. Because the compiler does not recognize the assembly code calls and variable modifications to C functions, the –pm option optimizes out those C functions. To keep these functions, place the FUNC_EXT_CALLED pragma (see section 5.6.4, *The FUNC_EXT_CALLED Pragma (–op*n *Option)*, on page 5-14) before any declaration or reference to a function that you want to keep.

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

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

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

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

Solution Compile with –pm –o3 –op2 to tell the compiler that outside functions do not call C functions or modify C variables. See section 3.3.1 for information about the –op2 option.

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

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

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

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

See section 3.3.1 for information about the –op*n* option.

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

Solution You *must* add the volatile keyword to the C variables that can be modified by the interrupts. Then you can optimize your code in one of these ways:

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

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

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

When you specify whole program compilation with the –pm option, you can use the –px *filename* option to specify the name of the output file. If you do not request assembly, the default file extension for the output file is .asm. If you request assembly, the default file extension for the output file is .obj. If you specify linking, you must name the output file with the –o option after the –z option or the name of the output file is the default name a.out.

Note: Symbolic Debugging and Optimized Code

If you use the –g compiler 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 compiler option; –mn reenables the optimizations disabled by –g.

3.4 Special Considerations When Using the Optimizer

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

3.4.1 Use Caution With asm Statements in Optimized Code

You must be extremely careful when using asm (inline assembly) statements in optimized code. The optimizer rearranges code segments, uses registers freely, and may completely remove variables or expressions. Although the compiler never optimizes out an asm statement (except when it is totally unreachable), the surrounding environment in which the assembly code is inserted may differ significantly from 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.

3.4.2 Use the Volatile Keyword for Necessary Memory Accesses

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 will not optimize out any references to volatile variables.

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

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

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

```
volatile unsigned int *ctrl
```

3.4.2.1 Use Caution When Accessing Aliased Variables

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 can 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 by writing through the pointer, but that it will not make its address available for use elsewhere after returning. For example, the called function cannot assign the local's address to a global variable or return it. In cases where this assumption is invalid, use the –ma compiler option to force the compiler to assume worst-case aliasing. In worst-case aliasing, any indirect reference (that is, using a pointer) can refer to such a variable.

3.4.2.2 Use the –ma Option to Indicate That the Following Technique Is Used

The optimizer assumes that any variable whose address is passed as an argument to a function will not be subsequently modified by an alias set up in the called function. Examples include:

- ❏ Returning the address from a function
- ❏ Assigning the address to a global

If you use aliases like this in your code, you must use the –ma option when you are optimizing your code. For example, if your code is similar to this, use the –ma option:

```
int *glob_ptr;

g()
{
    int  x = 1;
    int *p = f(&x);

    *p        = 5;    /* p aliases x    */
    *glob_ptr = 10;   /* glob_ptr aliases x */

    h(x);
}

int *f(int *arg)
{
    glob_ptr = arg;
    return arg;
}
```

3.5 Automatic Inline Expansion

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

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

The –oi*size* option controls only the inlining of functions that are not explicitly declared as inline. If you do not use the –oi option, the optimizer inlines very small functions. The –x shell option controls the inlining of functions declared as inline (see section 2.4.3.1, *Declaring a Function as Inline Within a Module*, on page 2-25).

3.6 Using the Interlist Utility With the Optimizer

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

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

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

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

Example 3–1 shows the function from Example 2–2 on page 2-28 compiled with the optimizer (–o2) and the –os option. The assembly file contains optimizer comments interlisted with assembly code.

Example 3–1. The Function From Example 2–2 Compiled With the –o2 and –os Options

```
FP  .set   AR2
    .global _main
    .global _a
    .bss
;   ac27 xx.c xx.if
;   opt27 –s –02 –z xx.if xx.opt
    .sect  ".text"
*************************************************************
;* FNAME: _main                    FR SIZE:  0               *
;*                                                           *
;* FUNCTION ENVIRONMENT                                      *
;*                                                           *
;* FUNCTION PROPERTIES                                       *
;*                               0 PARAMETER, 0 AUTO, 0 SOE  *
;*************************************************************

_main:
;***      ----------------   U$4 = &a[0];
;***      ----------------   L$1 = 9;
          MOV    AR4,#_a
          MOVB   AR6,#9
L2:
;***      ------------g2:
;*** 9    ---------------  *U$4++ = 0;
;*** 8    ---------------  if ( (--L$1) != (-1) ) goto g2;
;-------------------------------------------------------------
          MOV    *AR4++,#0              ; |9|
          BANZ   L2,AR6--               ; |8|
          ; branch occurs ; |8|
;***      ----------------------return;
          LRET
          ; return occurs
```

When you use the –ss and –os options with the optimizer, the optimizer inserts its comments and the interlist utility runs between the code generator and the assembler, merging the original C source into the assembly file.

Example 3–2. The Function From Example 2–2 Compiled With the –o2, –os, and –ss Options

```
FP  .set   AR2
    .global _main
    .global _a
    .bss
;   ac27 xx.c xx.if
;   opt27 –s –02 –z xx.if xx.opt
    .sect  ".text"
;---------------------------------------------------------------------------
;   5 | void main (void)
;   7 | int i;
;   8 | for ( i = o; i < 10; i++ )
;---------------------------------------------------------------------------
;***************************************************************************
;* FNAME: _main                                      FR SIZE:  0             *
;*                                                                           *
;* FUNCTION ENVIRONMENT                                                      *
;*                                                                           *
;* FUNCTION PROPERTIES                                                       *
;*                                                0 PARAMETER, 0 AUTO, 0 SOE  *
;***************************************************************************
_main:
;* AR6    assigned to L$1
;* AR4    assigned to U$4
;***      ----------------   U$4 = &a[0];
;***      ----------------   L$1 = 9;
          MOV    AR4,#_a              ; |4|
          MOVB   AR6,#9               ; |4|

L2:
;***      --------------g2:
;*** 9    --------------     *U$4 = &a[0];
;*** 8    --------------     if ( (--L$1) != (-1) ) goto g2;
;---------------------------------------------------------------------------
; 9 | a [i] = 0;
;---------------------------------------------------------------------------

          MOV    *AR4++,#0            ; |9|
          BANZ   L2,AR6--             ; |8|
          ; branch occurs ; |8|
;***      ----------------------return;
          LRET
          ; return occurs
```

3.7 Debugging Optimized Code

Debugging optimized code is not recommended, because the optimizer's extensive rearrangement of code and the many-to-many allocation of variables to registers often make it difficult to correlate source code with object code. To remedy this problem, you can use the –g and –o options to optimize your code in such a way that you can still debug it.

To debug optimized code, use the –g and –o options. The –g option generates symbolic debugging directives that are used by the C source debugger, but it disables many code generator optimizations. When you use the –o option (which invokes the optimizer) with the –g option, you turn on the maximum amount of optimization that is compatible with debugging.

If you want to use symbolic debugging and still generate fully optimized code, use the –mn option. The –mn option reenables the optimizations disabled by –g. However, if you use the –mn option, portions of the debugger's functionality will be unreliable.

3.8 What Kind of Optimization Is Being Performed?

The TMS320C27xx C compiler uses a variety of optimization techniques to improve the execution speed of your C programs and to reduce their size. Optimization occurs at various levels throughout the compiler.

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

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

Optimization	Page
Cost-based register allocation	3-18
Alias disambiguation	3-19
Data flow optimizations ❑ Copy propagation ❑ Common subexpression elimination ❑ Redundant assignment elimination	3-20
Expression simplification	3-20
Inline expansion of run-time-support library functions	3-21
Induction variable optimizations and strength reduction	3-22
Loop-invariant code motion	3-23
Loop rotation	3-23
Register variables	3-23
Register tracking/targeting	3-23

3.8.1 Cost-Based Register Allocation

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

In the following example an auxiliary register is allocated to the counting variable i. This allows the use of the BANZ instruction which implements the counting loop. Strength reduction turns the array references into efficient pointer references with auto increments.

Example 3–3. Strength Reduction, Induction Variable Elimination, Register Variables

(a) C Source

```
int a[10];

main ()
{
    int i;
    for (i = 0; i < 10; i++)
         a[i] = 0
}
```

Example 3–3. Strength Reduction, Induction Variable Elimination, Register Variables (Continued)

(b) Compiler output:

```
FP     .set      AR2
       .global   _main
       .global   _a
       .bss      _a,10,1,0
;      ac27 xx.c xx.if
;      opt27 –s –02 –z xx.if xx.opt
       .sect     ".text"

;*********************************************************
;* FNAME: _main                          FR SIZE:  0       *
;*                                                         *
;* FUNCTION ENVIRONMENT                                    *
;*                                                         *
;* FUNCTION PROPERTIES                                     *
;*                          0 Parameter, 0 Auto, 0 SOE     *
;*********************************************************

_main:
;***   ----------------------U$4 + &a[0];
;***   ----------------------L$1 = 9;
       MOV       AR4,#_a
       MOVB      AR6,#9

L2:
;***   ----------------------g2:
;*** 9 ----------------------U$4++ = 0;
;*** 8 ----------------------if ( (--L$1 != (-1) ) goto g2;
       MOV       *AR4++, #0                    ; |9|
       BANZ      L2,AR6--                      ; |8|
       ; branch occurs ; |8|
;***   ----------------------return;
       LRET
       ; return occurs
```

3.8.2 Alias Disambiguation

C programs 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.

3.8.3 Data Flow Optimizations

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

- **Copy propagation**

  Following an assignment to a variable, the compiler replaces references to the variable with its value. The value can be another variable, a constant, or a common subexpression. This can result in increased opportunities for constant folding, common subexpression elimination, or even total elimination of the variable (see Example 3–4 on page 3-21).

- **Common subexpression elimination**

  When two or more expressions produce the same value, the compiler computes the value once, saves it, and reuses it.

- **Redundant assignment elimination**

  Often, copy propagation and common subexpression elimination optimizations result in unnecessary assignments to variables (variables with no subsequent reference before another assignment or before the end of the function). The optimizer removes these dead assignments (see Example 3–4).

3.8.4 Expression Simplification

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

In Example 3–4, 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 plus multiplying j by 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.

Example 3–4. Data Flow Optimizations and Expression Simplification

(a) C source

```
int simplify (int j)
{
   int a = 3;
   int b = (j * a) + (j * 2);
   return b;
}
```

(b) Compiler output

```
FP  .set   AR2
    .global _simplify
;   ac27 xx.c xx.if
;   opt27 –s –02 –z xx.if xx.opt
    .sect     ".text"

;***************************************************
;* FNAME: _simplify                FR SIZE: 0      *
;*                                                 *
;* FUNCTION ENVIRONMENT                            *
;*                                                 *
;* FUNCTION PROPERTIES                             *
;*                          0 Parameter, 0 Auto, 0 SOE *
;***************************************************

_simplify:
;*** 5    ----------------------     return j*5;
          MOV    T,AL                   ; |5|
          MPYB   ACC,T,#5               ; |5|
          LRET
; return occurs
```

3.8.5 Inline Expansion of Run-Time-Support Library Functions

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

In Example 3–5, the compiler finds the code for the C function plus( ) and replaces the call with the code.

Example 3–5. Inline Function Expansion

(a) C source

```
int plus (int x, int y)
{
   return x+y;
}

main ()
{
   int a = 3;
   int b = 4;
   int c = 5;

   return plus (a, plus(b,c));
}
```

(b) Compiler output

```
;****************************************************
;* FNAME: _main                        FR SIZE:   0         *
;*                                                          *
;* FUNCTION ENVIRONMENT                                     *
;*                                                          *
;* FUNCTION PROPERTIES                                      *
;*                          0 Parameter, 0 Auto, 0 SOE      *
;****************************************************

_main:
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>ENTERING plus()
;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<LEAVING plus()
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>ENTERING plus()
;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<LEAVING plus()
;*** 12   ----------------------return 12;
          MOVB   AL,#12               ; |12|
          LRET
          ;return occurs
```

3.8.6 Induction Variables and Strength Reduction

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 of for loops are very often induction variables.

Strength reduction is the process of replacing inefficient 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.

Induction variable analysis and strength reduction together often remove all references to your loop-control variable, allowing its elimination).

3.8.7 Loop-Invariant Code Motion

This optimization identifies expressions within loops that always compute to 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.

3.8.8 Loop Rotation

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

3.8.9 Register Variables

The compiler helps maximize the use of registers for storing local variables, parameters, and temporary values. Accessing variables stored in registers is more efficient than accessing variables in memory. Register variables are particularly effective for pointers (see Example 3–3 on page 3-18).

3.8.10 Register Tracking/Targeting

The compiler tracks the contents of registers to avoid reloading values if they are used again soon. Variables, constants, and structure references such as (a.b) are tracked through straight-line code. Register targeting also computes expressions directly into specific registers when required, as in the case of assigning to register variables or returning values from functions (see Example 3–6 on page 3-24).

Example 3–6. Register Tracking/Targeting

(a) C source

```
int x, y;

main()
{
   x *= 3;
   y = x;
}
```

(b) Compiler output

```
FP  .set   AR2
    .global _main
    .global _x
    .bss    _x,1,1,0
;   ac27 xx.c xx.if
;   opt27 –02, –z xx.if, xx.opt
    .sect      ".text"

;***************************************************************
;* FNAME: _main                                FR SIZE: 0      *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                            0 Parameter, 0 Auto, o SOE       *
;***************************************************************

_main:
          MOV       DP,#_x
          MOV       T,@_x         ; |7|
          MPYB      ACC,T,#3      ; |7|
          MOV       @_y,AL        ; |8|
          MOV       @_x,AL        ; |7|
          LRET
          ;return occurs
```

Chapter 4

Linking C Code

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

- ❑ You can compile individual modules and then link them together.
- ❑ You can compile and link in one step by using cl27.

This chapter describes how to invoke the linker with each method. It also discusses special requirements of linking C code, including linking with a run-time-support library, specifying the type of initialization, and allocating the program into memory. For a complete description of the linker, see the *TMS320C27xx Assembly Language Tools User's Guide*.

Topic **Page**

4.1 Invoking the Linker as an Individual Program

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

lnk27 {**–c** | **–cr**} *filenames* [*–options*] [**–o** *name.out*] [**lnk.cmd**]

lnk27	The command that invokes the linker.
–c | **–cr**	Options that tell the linker to use special conventions defined by the C environment. When you use lnk27, you must use –c or –cr. The –c option uses automatic variable initialization at run time; the –cr option uses automatic variable initialization at load time.
options	Options affect how the linker handles your object files. Options can appear anywhere on the command line or in a linker command file. (Options are discussed in section 4.3, *Linker Options*.)
filenames	Names of object files, linker command files, or archive libraries. The default extension for all input files is *.obj*; any other extension must be explicitly specified. The linker can determine whether the input file is an object or ASCII file that contains linker commands. The default output filename is *a.out*, unless you use the –o option to name the output file.
–l *libraryname*	If you are linking C code, you must use a run-time-support library. You can use the libraries included with the compiler, or you can create your own run-time-support library. If you have specified a run-time-support library in a linker command file, you do not need this parameter. The –l option tells the linker to look outside the current file directory. See section 4.4.1, *Linking With Run-Time-Support Libraries*, on page 4-6.

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

```
lnk27 –c prog1 prog2 prog3 –o prog.out rts.lib
```

The linker uses a default allocation algorithm to allocate your program into memory. You can use the MEMORY and SECTIONS directives in the linker command file to customize the allocation process. For more information, see the *TMS320C27xx Assembly Language Tools User's Guide*.

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

The options and parameters discussed in this section apply to both methods of linking; however, when you link while compiling, the linker options must follow the –z option (see section 2.1.1, *Invoking the C Compiler Shell,* on page 2-4).

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

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

All arguments that follow –z on the command line are passed 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:

```
cl27 –sq *.c –z c.cmd –o prog.out rts.lib
```

First, all of the files in the current directory that have a *.c* extension are compiled using the –s (interlist C and assembly code) and –q (run in quiet mode) options. Second, the linker links the resulting object files by using the c.cmd command file. The –o option names the output file. Finally, the run-time-support library, rts.lib, is used to resolve any undefined references to run-time support functions.

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

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

4.2.1 Disabling the Linker (–c Shell Option)

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

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

4.3 Linker Options

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

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

–ar Produces a relocatable, executable object module

–b Disables merge of symbolic debugging information

–c Autoinitializes variables at run time

–cr Autoinitializes variables at load time

–plink Invokes the post-link optimizer, which modifies assembly language instructions to generate better code. For more information on –plink and the post-link optimizer, see the *TMS320C27xx Assembly Language Tools User's Guide*.

–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 null areas within output sections. *fill_value* is a 16-bit constant.

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

–h Makes all global symbols static

–heap *size* Sets the heap size (for dynamic memory allocation) to *size* bytes and defines a global symbol that specifies the heap size. The default is 0x400 bytes.

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

–j Disables conditional linking that has been set up with the assembler .clink directive

–k Forces the linker to ignore all alignment parameters specified in SECTIONS directives

–l *filename*	(Lower case L) Controls where the linker looks for the library
–m *filename*	Produces a map or listing of the input and output sections, including null areas, and places the listing in *filename.* The filename must follow operating-system conventions.
–n	Ignores all fill specifications in memory directives. Use this option in the development stage of a project to avoid generating large *.out* files, which can result from using memory-directive fill specifications.
–o *filename*	Names the executable output module. The *filename* must follow operating system conventions. If the –o option is not used, the default filename is a.out.
–q	Requests a quiet run (suppresses the banner)
–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. The default is 1K words.
–u *symbol*	Places the unresolved external symbol *symbol* into the output module's symbol table
–w	Displays a message when an undefined output section is created
–x	Forces rereading of libraries and resolves back references

For more information on linker options, see the linker description in the *TMS320C27xx Assembly Language Tools User's Guide.*

4.4 Controlling the Linking Process

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

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

This section discusses how these factors are controlled and provides a sample linker command file.

For more information about how to operate the linker, see the linker description in the *TMS320C27xx Assembly Language Tools User's Guide.*

4.4.1 Linking With run-time-Support Libraries

You must link all C programs with a run-time-support library. The library contains standard C functions as well as functions used by the compiler to manage the C environment. You can use the library included with the compiler, or you can create your own run-time-support library.

To specify a library as linker input, simply enter the library filename as you would any other input filename; the linker looks for the filename in the current directory. If you want to use a library that is not in the current directory, use the –l (lowercase L) linker option. To use the –l linker option, type on the command line using the following syntax:

lnk27 {**–c** | **–cr**} *filenames* **–l** *libraryname*

The –l option also tells the linker to look at the –i options and then the C_DIR environment variable to find an archive path or object file. For more information, see the *TMS320C27xx Assembly Language Tools User's Guide.*

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

You must link all C programs with an object module called *boot.obj*. When a C program begins running, it must execute boot.obj first. The boot.obj file contains code and data to initialize the runtime environment; the linker automatically extracts boot.obj and links it when you use –c or –cr and include rts27.lib in the link.

Note: The _c_int00 Symbol

One important function contained in the run-time-support library is _c_int00. The symbol _c_int00 is the starting point in boot.obj; if you use the –c or –cr linker option, _c_int00 is automatically defined as the entry point for the program. If your program begins running from load time, you should set up the loadtime vector to branch to _c_int00 so that the processor executes boot.obj first.

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

1) Sets up the stack

2) Processes the runtime initialization table and autoinitializes global variables (when using the –c option)

3) Calls main

4) Calls exit when main returns

Chapter 7, *Run-Time-Support Functions*, describes additional run-time-support functions that are included in the library. These functions include ANSI C standard runtime support.

4.4.2 Specifying the Type of Initialization

The C compiler produces data tables for autoinitializing global variables. Section 6.8.3, *Initialization Tables*, on page 6-30 discusses the format of these tables. These tables are in a named section called *.cinit*. The initialization tables are used in one of the following ways:

- Autoinitializing variables at run time. Global variables are initialized at *run time*. Use the –c linker option (see section 6.8.4, *Autoinitialization of Variables at Run Time*, on page 6-32).

- Autoinitializing variables at load time. Global variables are initialized at *load time*. Use the –cr linker option (see section 6.8.5, *Autoinitialization of Variables at Load Time*, on page 6-33).

When you link a C program, you must use either the –c or –cr linker option. These options tell the linker to select autoinitialization at run time or load time. When you compile and link programs using the compiler shell, the –c linker option is the default. (If specified, the –c linker option must follow the –z option, see section 4.2, *Invoking the Linker With the Compiler Shell (–z Option)*, on page 4-3.) The following list outlines the linking conventions used with –c or –cr:

- The symbol _c_int00 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_int00 is automatically referenced, ensuring that boot.obj is linked in from the run-time-support library.
- The *.cinit* output section is padded with a termination record so that the loader (loadtime initialization) or the boot routine (runtime initialization) knows when to stop reading the initialization tables.
- When using autoinitialization at load time (the –cr linker option), the following occur:
  - The linker sets the symbol *cinit* to –1. This indicates that the initialization tables are not in memory, so no initialization is performed at run time.
  - The STYP_COPY flag (010h) is set in the .cinit section header. STYP_COPY is the special attribute that tells the loader to perform autoinitialization directly and not to load the .cinit section into memory. The linker does not allocate space in memory for the .cinit section.
- When autoinitializing at run time (–c linker option), the linker defines the symbol *cinit* as the starting address of the .cinit section. The boot routine uses this symbol as the starting point for autoinitialization.

4.4.3 Specifying Where to Allocate Sections in Memory

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

The compiler creates two basic kinds of sections: initialized and uninitialized. Table 4–1 summarizes the sections.

Table 4–1. Sections Created by the Compiler

(a) Initialized sections

Name	Contents
.cinit	Tables for explicitly initialized global and static variables
.const	Global and static const variables that are explicitly initialized and that are string literals
.switch	Tables for implementing switch statements.
.text	Executable code and constants

(b) Uninitialized sections

Name	Contents
.bss	Global and static variables
.stack	Stack
.sysmem	Memory for malloc functions

When you link your program, you must specify where to allocate the sections in memory. In general, initialized sections are linked into ROM or RAM; uninitialized sections are linked into RAM. See section 6.1.1, *Sections*, on page 6-3 for a complete description of how the compiler uses these sections. The linker provides MEMORY and SECTIONS directives for allocating sections. For more information about allocating sections into memory, see the linker description in the *TMS320C27xx Assembly Language Tools User's Guide*.

4.4.4 A Sample Linker Command File

Example 4–1 shows a typical linker command file that links a C program. The command file in this example is named lnk.cmd. It links three object files (a.obj, b.obj, and c.obj) and creates a program (prog.out) and a map file (prog.map).

To link the program, enter:

```
lnk27 lnk.cmd
```

The MEMORY and SECTIONS directives, may require modification to work with your system. See the linker description chapter in the *TMS320C27xx Assembly Language Tools User's Guide* for information on these directives.

Example 4–1. Linker Command File

```
a.obj b.obj c.obj           /* Input filenames       */
–c prog.out –m prog.map     /* Options               */

MEMORY                      /* MEMORY directive      */
{
  RAM:  origin = 100h      length = 0100h
  ROM:  origin = 01000h    length = 0100h
}

SECTIONS                    /* SECTIONS directive    */
{
  .text:   > ROM
  .data:   > ROM
  .bss:    > RAM
  .cinit:  > ROM
  .switch: > ROM
  .const:  > RAM
  .stack:  > RAM
  .sysmem: > RAM
}
```

Chapter 5

TMS320C27xx C Language Implementation

The C language that the TMS320C27xx 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.

Topic **Page**

5.1 Characteristics of TMS320C27xx C

The ANSI standard identifies some features of the C language that are affected by characteristics of the target processor, runtime environment, or host environment. For reasons of efficiency or practicality, this set of features may differ among standard compilers. This section describes how these features are implemented for the ’C27xx C compiler.

The following list identifies all such cases and describes the behavior of the ’C27xx C compiler in each case. Each description also includes a reference to the formal ANSI standard and to *The C Programming Language* (second edition) by Kernighan and Ritchie (K&R).

5.1.1 Identifiers and Constants

The following conventions apply for identifiers and constants:

- The first 100 characters of all identifiers are significant. Case is significant; uppercase and lowercase characters are distinct for identifiers. These characteristics apply to all identifiers, internal and external, in all TMS320C27xx 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)

- Hexadecimal or octal escape sequences in character or string constants may have values up to 32 bits. (ANSI 3.1.3.4, K&R A2.5.2)

- Character constants with multiple characters are encoded as the last character in the sequence. For example,

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

5.1.2 Data Types

The following conventions apply for data types:

- For information about the representation of data types, see section 5.2, *Data Types*, on page 5-5. (ANSI 3.1.2.5, K&R A4.2)

- The type size_t, which is 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)

5.1.3 Conversions

The following conventions apply for conversions:

- Float-to-integer conversions truncate toward 0. (ANSI 3.2.1.3, K&R A6.3)
- Pointers and integers can be freely converted. (ANSI 3.3.4, K&R A6.6)

5.1.4 Expressions

The following conventions apply for 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)

5.1.5 Declarations

The following conventions apply for 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 (or 32-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. Therefore, bit fields are limited to a maximum size of 16 bits, regardless of what size is used in the C source. (ANSI 3.5.2.1, K&R A8.3)

5.1.6 Preprocessor

The preprocessor recognizes the following pragma directives; all others are ignored. Pragma directives tell the compiler's preprocessor how to treat functions. The recognized pragmas are:

- ❏ CODE_SECTION (*func, "section name*)
- ❏ DATA_SECTION (*symbol, "section name"*)
- ❏ INTERRUPT (*func*)
- ❏ FUNC_EXT_CALLED

(ANSI 3.8.6, K&R A12.8)

For more information on pragmas, see section 5.6, *Pragma Directives*, on page 5-10.

5.1.7 Header Files

The following applies to header files. For detailed information about header files, see section 7.3, *Header Files*, on page 7-13.

- ❏ The following ANSI C run-time support functions are not supported:
  - locale.h
  - signal.h
  - time.h

(ANSI 4.1, K&R B)

- ❏ The stdlib library functions getenv and system are not supported.

(ANSI 4.10.4, K&R B5)

- ❏ For functions in the math library that produce a floating-point return value, if the values are too small to be represented, zero is returned and errno is set to ERANGE. See sections 7.3.3, *Error Reporting*, on page 7-15, and 7.3.6, *Floating-Point Math*, on page 7-18, for information on error reporting and floating-point math.

5.2 Data Types

The following information applies to 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; in expressions, the type enum is equivalent to int.
- All floating-point types (float, double, and long double) are equivalent and are represented as IEEE single-precision format.

The size, representation, and range of each scalar data type are listed in the table below.

Many of the range values are available as standard macros in the header file limits.h, which is supplied with the compiler. For more information, see section 5.9, *Limits (float.h and limits.h)*, on page 5-18.

Table 5–1. TMS320C27xx 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	IEEE 32-bit	1.19209290e–38	3.4028235e+38
double	32 bits	IEEE 32-bit	1.19209290e–38	3.4028235e+38
long double	32 bits	IEEE 32-bit	1.19209290e–38	3.4028235e+38
pointers	16 bits	Binary	0	0xFFFF

Note: ’C27xx 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 ’C27xx char is 16 bits (to make it separately addressable), a byte is also 16 bits. This yields results you may not expect; for example, size of (int) = = 1 (not 2). ’C27xx bytes and words are equivalent (16 bits).

5.3 Register Variables

The C compiler treats register variables (variables declared with the register keyword) differently, depending on whether you use the optimizer.

- **Compiling with the optimizer**

  The compiler ignores any register declarations and allocates registers to variables and temporary values by using a cost algorithm that attempts to make the most efficient use of registers.

- **Compiling without the optimizer**

  If you use the register keyword, you can suggest variables as candidates for allocation into registers. The same set of registers used for allocation of temporary expression results is used for allocation of register variables.

The compiler attempts to honor all register definitions. If the compiler runs out of appropriate registers, it frees a register by moving its contents to memory. If you define too many objects as register variables, you limit the number of registers the compiler has for temporary expression results. The limit causes excessive movement of register contents to memory.

Any object with a scalar type (integer, floating point, or pointer) can be declared as a register variable. The register designator is ignored for objects of other types.

The register storage class is meaningful for parameters as well as local variables. Normally, in a function, some of the parameters are copied to a location on the stack where they are referenced during the function body. A register parameter is copied to a register instead of the stack. This action speeds access to the parameter within the function.

For more information on register variables, see section 6.2, *Register Conventions*, on page 6-9.

5.4 The asm Statement

The ’C27xx C compiler allows you to imbed ’C27xx 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 a single 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 specify 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 is detected by the assembler. For more information on assembly statements, refer to the *TMS320C27xx Assembly Language Tools User’s Guide*.

The asm statements do not follow the syntactic restrictions of normal C statements. Each can appear as either a statement or a declaration, even outside code 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 compiler cannot remove asm statements (except where such statements are totally unreachable), it can significantly rearrange the code order near asm statements, possibly causing undesired results. The asm command is provided so that you can access hardware features, which, by definition, C is unable to access.

5.5 The interrupt Keyword

The 'C27xx compiler extends the C language by adding the *interrupt* keyword to specify that a function is to be treated as an interrupt function.

Functions that handle interrupts follow special register-saving rules and a special return sequence. When C code is interrupted, the interrupt routine must preserve the contents of all machine registers that are used by the routine or by any function called by the routine. When you use the interrupt keyword with the definition of the function, the compiler generates register saves based on the rules for interrupt functions and the special return sequence for interrupts.

You can only use the interrupt keyword with a function that is defined to return void and that has no parameters. The body of the interrupt function can have local variables and is free to use the stack or global variables. For example:

```
interrupt void int_handler()
{
    unsigned int flags;

    ...
}
```

The name c_int00 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_int00 does not save any registers.

5.6 Pragma Directives

Pragma directives tell the compiler's preprocessor how to treat functions. The 'C27xx C compiler supports the following pragmas:

- ❏ CODE_SECTION (*func, "section name"*)
- ❏ DATA_SECTION (*symbol, "section name"*)
- ❏ INTERRUPT (*func*)
- ❏ FUNC_EXT_CALLED (*func*)

The arguments *func* and *symbol* cannot defined or declared inside the body of a function. You must specify the pragma outside the body of a function; and it must occur before any declaration, definition, or reference to the *func* or *symbol* argument. If you do not do this, the compiler issues a warning.

5.6.1 The CODE_SECTION Pragma

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

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

#pragma CODE_SECTION (*func*, *"section name"***)**

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

Example 5–1. Using the CODE_SECTION Pragma

(a) C source file

```
char bufferA[80];
char bufferB[80];

#pragma CODE_SECTION(funcA, ”codeA”)

char funcA(int i);
char funcB(int i);

void main()
{
   char c;
   c = funcA(1);
   c = funcB(2);
}

char funcA (int i)
{
   return bufferA[i];
}

char funcB (int j)
{
   return bufferB[j];
}
```

Example 5–1.Using the CODE_SECTION Pragma (Continued)

(b) Assembly source file

```
_main:
        ADDB      SP,#2
        MOVB      AL,#1
        LC        #_funcA
        ; call occurs [#_funcA]
        MOV       *–SP[1],AL
        MOVB      AL,#2
        LC        #_funcB
        ; call occurs [#_funcB]
        MOV       *–SP[1],AL
        SUBB      SP,#2
        LRET
        ; branch occurs
        .sect     ”codeA”
;***************************************************************
;* FNAME: _funcA                          FR SIZE:   1       *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                            0 Parameter,  1 Auto,  0 SOE *
;***************************************************************
_funcA:
        INC       SP
        MOV       *–SP[1],AL
        MOV       AR3,#_bufferA
        MOV       AR0,*–SP[1]
        MOV       AL,*+AR3[AR0]
        DEC       SP;
        ;branch occurs .sect”.text”

;***************************************************************
;* FNAME: _funcB                          FR SIZE:   1       *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                           0 Parameter,  1 Auto,  0 SOE  *
;***************************************************************
_funcB:
        INC       SP
        MOV       *–SP[1],AL
        MOV       AR3,#_bufferB
        MOV       AR0,*–SP[1]
        MOV       AL,*+AR3[AR0]
        DEC       SP
        LRET
        ;branch occurs
```

5.6.2 The DATA_SECTION Pragma

The DATA_SECTION pragma allocates space for the *symbol* in a section called *section name*.

The syntax for the pragma is:

#pragma DATA_SECTION (*symbol*, "*section name'***)**

The DATA_SECTION pragma is useful if you have data objects that you want to link into an area separate from the .bss section. Example 5–2 demonstrates the use of the DATA_SECTION pragma.

Example 5–2. Using the DATA_SECTION Pragma

(a) C source file

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

(b) Assembly source file

```
          .global _bufferA
          .bss    _bufferA,512,1
          .global _bufferB
_bufferB: .usect  "my_sect",512,1
```

5.6.3 The INTERRUPT Pragma

The INTERRUPT pragma allows you to handle interrupts directly with C code. The argument *func* is the name of a function. The syntax of the pragma is:

#pragma INTERRUPT (*func***)**

5.6.4 The FUNC_EXT_CALLED Pragma

When you use the –pm and –o3 options, the compiler uses program-level optimization. When you use this type of optimization, the compiler removes any function that is not called, directly or indirectly, by main. You can have C functions that are called by hand-coded assembly instead of main. The FUNC_EXT_CALLED pragma allows you to instruct the compiler to keep these C functions or any other functions that these C functions call. These functions act like main; they function as entry points into C.

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

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

The argument *func* is the name of the C function that is called by hand-coded assembly.

Except for _c_int00, which is the name reserved for the system reset interrupt for C programs, the name of the interrupt (the *func* argument) does not need to conform to a naming convention.

5.7 Initializing Static and Global Variables

The ANSI C standard specifies that static and global (extern) variables without explicit initializations must be initialized to zero 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 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 or sections created using the data pragma:

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

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

5.7.1 Initializing Static and Global Variables With the Const Type Qualifier

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

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 in section 5.7, *Initializing Static and Global Variables*.). For example:

```
const int zero;       /* may not be initialized to 0     */
```

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 0       */
```

which corresponds to an entry in the .const section:

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

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

5.8 Compatibility with K&R C

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

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

To simplify the process of compiling existing C programs with the 'C27xx ANSI C compiler, the compiler has a K&R option (–pk) that modifies some semantic rules of the language for compatibility with older code. In general, the –pk option relaxes requirements that are stricter for ANSI C than for K&R C. The –pk option does not disable any new features of the language such as function prototypes, enumerations, initializations, or preprocessor constructs. Instead, –pk simply liberalizes the ANSI rules without revoking any of the features.

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

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

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

- ANSI prohibits 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. The –pe option, 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        /* warning unless –pk used */
```

5.9 Compiler Limits

Due to the variety of host systems supported by the 'C27xx 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 avoid 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 5–2 specifies the limits that are absolute. All the absolute limits equal or exceed those required by the ANSI C standard.

Table 5–2. Absolute Compiler Limits

Description	Limits
Filename length	512 characters
Source line length; reflects the number of characters after splicing of \ lines; applies to any single macro definition or invocation	16K characters
Length of strings built from # or ##; reflects the number of characters before concatenation; all other character strings are unrestricted.	512 characters
Macro definitions	Allocated from available system memory
Macros predefined with –d	64
Macro parameters	32 parms
Macro nesting; includes argument substitutions.	32 levels
#include search paths; includes –i and C_DIR directories	64 paths
#include file nesting	64 levels
Conditional inclusion (#if) nesting	64 levels
Nesting of struct, union, or prototype declarations	20 levels
Function parameters	48 parms
Array, function, or pointer derivations on a type	12 derivations
Aggregate initialization nesting	32 levels
Static initializers	1500 per initialization (approximately)
Local initializers	150 levels (approximately)
Nesting of declarations in structures, unions, or protoypes	32 levels
Global symbols; may be further limited by available system memory	10000
Block scope symbols visible at any point	1000
Number of unique string constants	1000
Number of unique floating-point constants	1000

Chapter 6

Runtime Environment

This chapter describes the TMS320C27xx C runtime environment. To ensure successful execution of C programs, it is critical that all runtime code maintain this environment. Without the proper environment, your code is not reliable. It is also important to follow the guidelines in this chapter if you write assembly language functions that interface with C code.

Topic **Page**

6.1 Memory Model

The 'C27xx treats memory as two linear blocks of program memory and data memory:

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

Blocks of code or data generated by a C program are placed into contiguous blocks in the appropriate memory space.

Note: Linker Information

1) **Link Sections in the Low 64K of Data Memory** – Although the TMS320C27xx device can address 22-bits of data, the preliminary release of the compiler only accesses 16-bits (the low 64K) of data memory.

   It is required that the .bss (and any user-defined sections that will hold data) must be linked in the low 64K of data memory.

2) **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).

6.1.1 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 sections, see the *TMS320C27xx 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 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 run time 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 can 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 used by malloc functions. If no malloc functions are used, the size of the section remains 0.

The linker takes the individual sections from different modules and combines sections with the same name to create output sections. The complete program is made up of these output sections. 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 must be linked into RAM and must be in data memory. 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, 1
.const	ROM or RAM	1
.bss	RAM	1
.stack	RAM	1
.sysmem	RAM	1

For more information about allocating sections into memory, see the introductory COFF information in the *TMS320C27xx Assembly Language Tools User's Guide*.

6.1.2 C System Stack

The C compiler uses a stack to:

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

The runtime stack grows up from low addresses to higher addresses. By default, it is allocated in the .stack section. (See the rts file, boot.asm.) The compiler uses the hardware stack pointer (SP) to manage the stack.

For frames that exceed 63 words in size (the maximum reach of the SP offset addressing mode), the compiler uses AR2 as a frame pointer (FP). Each function invocation creates a new frame at the top of the stack, from which local and temporary variables are allocated. The FP points at the beginning of this frame to access memory locations that can not be referenced directly using the SP.

Note: Linking .stack Section

The .stack section has to be linked into the low 64K of data memory. The SP is a 16-bit register and cannot access addresses beyond 64K.

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 words. You can change the size of the stack at link time by using the –stack linker command option.

Note: Stack Overflow

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

6.1.3 Allocating .const to Program Memory

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

Modify the boot routine:

1) Extract boot.asm from the source library:

```
ar27 –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:

```
asm27 boot.asm
```

4) Archive the boot routine into the object library:

```
ar27 –r rts.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;
               }
   ...
}
```

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

6.1.4 Dynamic Memory Allocation

The run-time-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 run time. 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 run-time-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 words. 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.

6.1.5 Initialization of Variables

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

In situations where a program is loaded directly from an object file into memory and 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 run time). You can specify this *to the linker* by using the –cr linker option. For more information on system initialization, see section 6.8, *System Initialization*, on page 6-29.

6.1.6 Allocating Memory for Static and Global Variables

A unique, contiguous space is allocated for all external or static variables 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.

6.1.7 Field/Structure Alignment

When the compiler allocates space for a structure, it allocates as many words as are needed to hold all of the structure's members and comply with alignment constraints for each member.

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.

6.1.8 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, see section 6.8, *System Initialization*, on page 6-29.

- 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 included. For example, the following lines define the string abc, along with the terminating byte; the label SL5 points to the string:

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

  String labels have the form SL*n*, where *n* is a number assigned by the compiler to make the label unique. These numbers begin at 0 with an increase of 1 for each defined string. 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 is not duplicated in memory. All uses of an identical string constant share a single definition of the string.

Because strings are stored in .const (possibly 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! */
```

6.2 Register Conventions

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

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

Table 6–1 summarizes how the compiler uses the 'C27xx registers and shows which registers are defined to be preserved across function calls.

Table 6–1. Register Use and Preservation Conventions

Register(s)	Usage	Save on Entry	Save on Call
AL	Expressions, argument passing, and returns 16-bit results from functions	No	Yes
AH	Expressions and argument passing	No	Yes
AR0	Pointers and expressions	No	Yes
AR1	Pointers and expressions	Yes	No
AR2	Pointers, expressions, and frame pointer (when needed)	Yes	No
AR3	Pointers and expressions	Yes	No
AR4	Pointers, expressions, argument passing, and returns pointer values from functions	No	Yes
AR5	Pointers, expressions, and arguments	No	Yes
XAR6	Pointers and expressions	No	Yes
XAR7	Pointers, expressions, indirect calls and branches (used to implement pointers to functions and switch statements)	No	Yes
SP	Stack pointer	†	†
T	Multiply and shift expressions	No	Yes
PL	Multiply expressions and Temp variables	No	Yes
PH	Multiply expressions and Temp variables	No	Yes
DP	Data page pointer (used to access global variables)	No	No

† The SP is preserved by the convention that everything pushed on the stack is popped off before returning.

6.2.1 Status Registers

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

Table 6–2. Status Register Fields

Field	Name	Presumed Value	Modified
SXM	Sign extension mode	–	Yes
TC	Test/control flag	–	Yes
C	Carry	–	Yes
Z	Zero flag	–	Yes
N	Negative flag	–	Yes
V	Overflow flag	–	Yes
PM	Product shift mode	0†	Yes
PAGE0	Direct/stack address mode	0†	No
SPA	Stack pointer align bit	–	Yes (in interrupts)

† The initialization routine that sets up the C run time environment will set these fields to the presumed value.

All other fields are not used and do not affect code generated by the compiler.

6.2.2 Register Variables

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

6.2.2.1 Register Variables When the Optimizer Is Not Used

When the optimizer is not used, the compiler attempts to allocate registers for variables declared with the register keyword.

6.2.2.2 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.

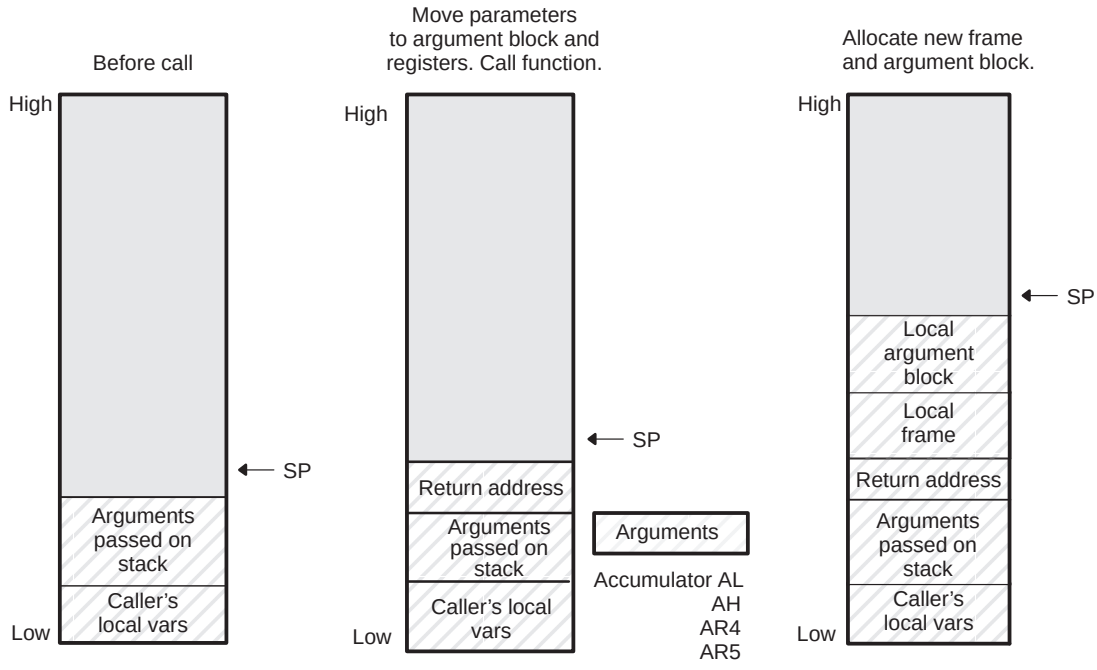
6.3 Function Calling Conventions

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

Figure 6–1 illustrates a typical function call. In this example, parameters that cannot be placed in registers are passed to the function on the stack. The function then allocates local variables and calls another function. This example shows the allocated local frame and argument block for the called function. Functions that have no local variables and do not require an argument block do not allocate a local frame.

The term *argument block* refers to the part of the local frame used to pass arguments to other functions. Parameters are passed to a function by moving them into the argument block rather then pushing them on the stack. The local frame and argument block are allocated at the same time.

Figure 6–1. Use of the Stack During a Function Call



6.3.1 How a Function Makes a Call

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

1) Any registers whose values are not necessarily preserved by the function being called (registers that are not save on entry (SOE)), but will be needed after the function returns are saved on the stack.

2) If the called function returns a structure, the calling function allocates the space for the structure and pass the address of that space to the called function as the first argument.

3) Arguments passed to the called function are placed in registers and, when necessary, placed on the stack.

   Arguments are placed in registers using the following scheme:

   a) If there are any 32-bit arguments (longs, floats, or pointers to functions) the first is placed in the 32-bit ACC (AH/AL). All other 32-bit arguments or function pointers are placed on the stack in reverse order.

   b) Pointer arguments are placed in AR4 and AR5. All other pointers are placed on the stack.

   c) Remaining 16-bit arguments are placed in the order AL, AH, AR4, AR5 if they are available.

4) Any remaining arguments not placed in registers are pushed on the stack in reverse order. That is, the leftmost argument that is placed on the stack is pushed on the stack last. All 32-bit arguments will be aligned to even addresses on the stack.

   A structure argument is passed as the address of the structure. The called function must make a local copy.

   For a function declared with an ellipsis, indicating that it is called with varying numbers of arguments, the convention is slightly modified. The last explicitly declared argument is passed on the stack so that its stack address can act as a reference for accessing the undeclared arguments.

   Some examples of function calls that show where arguments are placed are listed below:

```
func1 (int a, int b. long c) ;
       AR4    AR5    AH/AL
func1 (long a, int b, long c) ;
       AH/AL   AR4    stack
vararg (int a, int b, int c, ...)
        AL     AH     stack
```

5) The caller calls the function.

6.3.2 How a Called Function Responds

A called function performs the following tasks:

1) If the called function modifies AR1, AR2, or AR3, it must save them, since the calling function assumes that the values of these registers are preserved upon return. Any other registers may be modified without preserving them.

2) The called function allocates enough space on the stack for any local variables, temporary storage area, and arguments to functions that this function might call. This allocation occurs once at the beginning of the function by adding a constant to the SP register.

3) If the called function expects a structure argument, it receives a pointer to the structure instead. If writes are made to the structure from within the called function, space for a local copy of the structure must be allocated on the stack and the local structure must be copied from the passes pointer to the structure. If no writes are made to the structure, it can be referenced in the called function indirectly through the pointer argument.

   You must be careful to properly declare functions that accept structure arguments, both at the point where they are called (so that the structure argument is passed as an address) and at the point where they are declared (so the function knows to copy the structure to a local copy).

4) The called function executes the code for the function.

5) The called function returns a value. It is placed in a register using the following convention:

   16-bit integer value: AL
   32-bit integer value: ACC
   16-bit pointer: AR4

   If the function returns a structure, the caller allocates space for the structure and passes the address of the return space to the called function in AR4. To return a structure, the called function copies the structure to the memory block pointed by the extra argument.

   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(x), where S is a structure and F is a function that returns a structure, the caller can actually make the call as f(&s, x). The function f then copies the return structure directly into s, performing the assignment automatically.

   If the caller does not use the return structure value, an address value of 0 can be passed as the first argument. This directs the called function not to copy the return structure.

You must be careful to properly declare functions that return structures both at the point where they are called (so that the extra argument is passed) and at the point where they are declared (so the function knows to copy the result).

6) The called function deallocates the frame by subtracting the value that was added to the SP earlier.

7) The called function restores the values of all registers saved in step 1.

8) The called function returns.

6.3.3 Special Case for a Called Function (Large Frames)

If the space that needs to be allocated on the stack (step 2 in the previous section) is larger than 63 words, additional steps and resources are required to ensure that all local nonregister variables can be accessed. Large frames require using a frame pointer register (AR2) to reference local non-register variables within the frame. Prior to allocating space on the frame, the frame pointer is set up to point at the first argument on the stack that was passed on to the called function. If no incoming arguments are passed on to the stack, the frame pointer points to the return address of the calling function, which is at the top of the stack upon entry to the called function.

Avoid allocating large amounts of local data when possible. For example, do not declare large arrays within functions.

6.3.4 Accessing Arguments and Local Variables

A function accesses its local nonregister variables and its stack arguments indirectly through either the SP or the FP (frame pointer, designated to be AR2). All local and argument data that can be accessed with the SP use the *–SP [offset] addressing mode since the SP always points one past the top of the stack and the stack grows toward larger addresses.

Note: The PAGE0 Mode Bit Must Be Reset

Since the compiler uses the *–SP [offset] addressing mode, the PAGE0 mode bit *must be* reset (set to 0).

The largest offset available using *–SP [offset] is 63. If an object is too far away from the SP to use this mode of access, the compiler uses the FP (AR2). Since FP points at the bottom of the frame, accesses made with the FP use either *+FP [offset] or *+FP [AR0/AR1] addressing modes. Since large frames require utilizing AR2 and possibly an index register, extra code and resources are required to make local accesses.

6.3.5 Allocating the Frame and Accessing 32-Bit Values in Memory

Some ’C27xx instructions read and write 32 bits of memory at once (MOVL, ADDL, etc.). These instructions require that 32-bit objects be allocated on an even boundary. To ensure that this occurs, the compiler takes these steps:

1) It initializes the SP to an even boundary.

2) Because a call instruction adds 2 to the SP, it assumes that the SP is pointing at an even address.

3) It makes sure that the space allocated on the frame totals an even number, so that the SP points to an even address.

4) It makes sure that 32-bit objects are allocated to even addresses, relative to the known even address in the SP.

5) Because interrupts cannot assume that the SP is odd or even, it aligns the SP to an even address.

For more information on how these instructions access memory, see the *TMS320C27xx Assembly Language Tools User’s Guide*.

6.4 Interfacing C With Assembly Language

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

- ❏ Use separate modules of assembled code and link them with compiled C modules (see section 6.4.1, *Assembly Language Modules* on page 6-16). This is the most versatile method.
- ❏ Use inline assembly language embedded directly in the C source (see section 6.4.4, *Inline Assembly Language*, on page 6-21).
- ❏ Use intrinsics in C source to call an assembly language statement directly (see section 6.4.5, *Using Intrinsics to Access Assembly Language Statements*, on page 6-22).

6.4.1 Using Assembly Language Modules With C Code

Interfacing with assembly language functions is straightforward if you follow the calling conventions defined in section 6.3, *Function Calling Conventions*, on page 6-11, and the register conventions defined in section 6.2, *Register Conventions*, on page 6-9. C code can access variables and call functions defined in assembly language, and assembly code can access C variables and call C functions.

Follow these guidelines to interface assembly language and C:

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

  AR1
  AR2
  AR3
  SP

  If the SP is used normally, it does not need to be preserved explicitly. 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 function returns (thus preserving the SP).

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

- ❏ Interrupt routines must save *all* the registers they use. (For more information about interrupt handling, see section 6.5, *Interrupt Handling*, on page 6-24.)

- When you call a C function from assembly language, load the designated registers with arguments and push the remaining arguments onto the stack as described in section 6.3.1, *How a Function Makes a Call*, on page 6-12.

  When accessing arguments passed in from a C function, these same conventions apply.

- Longs and floats are stored in memory with the least significant word at the lower address.

- Structures are returned as described in step 5 in section 6.3.2, *How a Called Function Responds*, on page 6-13.

- 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 prepends 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 can 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 or .def 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 or .ref. This creates an undefined external reference that the linker resolves.

- Because compiled code runs with the PAGE0 mode bit reset, if you set the PAGE0 bit to 1 in your assembly language function, you must set it back to 0 before returning to compiled code.

Example 6–1 illustrates a C function called main, which calls an assembly language function called asmfunc. The asmfunc function takes its single argument, adds it to the C global variable called gvar, and returns the result.

Example 6–1. Calling an Assembly Language Function From C

(a) C program

```
extern int asmfunc(); /* declare external asm function  */
int gvar=4;           /* define global variable        */

main()
{
   int i;
   i=1;
   i = asmfunc(i);   /* call function normally        */
}
```

(b) Assembly language program

```
         .global _gvar
         .global _asmfunc
 _asmfunc:
         MOV      DP, #_gvar
         ADDB     AL, #5
         MOV      @_gvar, AL
         LRET
```

In the C program in Example 6–1, the extern declaration of asmfunc is optional, because the function returns an int. Like C functions, assembly functions need to be declared only if they return noninteger values. In the assembly language code in Example 6–1, note the underscores on all the C symbol names used in the assembly code.

The parameter i is passed in register AL.

6.4.2 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 6–2 shows an example that accesses a variable defined in .bss.

Example 6–2. 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 do not 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 6–3 shows an example that accesses a variable that is not defined in .bss.

Example 6–3. Accessing From C a Variable Not Defined in .bss

(a) C program

```
extern float sine[];  /* This is the object             */
float *sine_p = sine; /* Declare pointer to point to it */
f = sine_p[4];        /* Access sine as normal array    */
```

(b) Assembly language program

```
    .global  _sine       ; Declare variable as external
    .sect    "sine_tab"  ; Make a separate section
_sine:                   ; The table starts here
    .float   0.0
    .float   0.015987
    .float   0.022145
```

6.4.3 Accessing Assembly Language Constants

You can define global constants in assembly language by using the .set, .def, and .global directives, or you can define them in a linker command file using linker assignment statement. These constants are accessible from C only with the use of special operators.

For normal variables defined in C or assembly language, the symbol table contains *the address of the value* of the variable. For assembler constants, however, the symbol table contains the *value* of the constant. The compiler cannot tell which items in the symbol table are values and which are addresses.

If you try to access an assembler (or linker) constant by name, the compiler attempts to fetch a value from the address represented in the symbol table. To prevent this unwanted fetch, you must use the & (address of) operator to get the value. In other words, if x is an assembly language constant, its value in C is &x.

You can use casts and #defines to ease the use of these symbols in your program, as in Example 6–4.

Example 6–4. Accessing an Assembly Language Constant From C

(a) C program

```
extern int table_size; /*external ref */
#define TABLE_SIZE ((int) (&table_size))
       .            /* use cast to hide address–of */
       .
       .
for (i=0; i<TABLE_SIZE; ++i)
                    /* use like normal symbol */
```

(b) Assembly language program

```
_table_size   .set   10000         ; define the constant
              .global _table_size ; make it global
```

Since you are referencing only the symbol's value as stored in the symbol table, the symbol's declared type is unimportant. In Example [# from above], int is used. You can reference linker-defined symbols in a similar manner.

6.4.4 Inline Assembly Language

Within a C program, you can use the *asm statement* to insert 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 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");
```

6.4.5 Using Intrinsics to Access Assembly Language Statements

The 'C27xx compiler recognizes a number of intrinsic operators. Intrinsics are used like functions and produce assembly language statements that would otherwise be inexpressible in C. You can use C variables with these intrinsics, just as you would with any normal function.

The intrinsics are specified with two leading underscores, and are accessed by calling them as you do a function. For example:

```
long lvar;
int  ivar;
unsigned int uivar;
lvar = __mpyxu(ivar, uivar);
```

The compiler performs parameter checking if the header file c27xx.h is included in the source files that call intrinsics. The header file c27xx.h contains prototypes for the intrinsics.

The intrinsics listed in Table 6–3 are included. They correspond to the indicated 'C27xx assembly language instruction(s). See the *TMS320C27xx CPU and Instruction Set Reference Guide* for more information.

Table 6–3. TMS320C27xx C Compiler Intrinsics

C Compiler Intrinsic	Assembly Instruction	Description
long__**mpyxu(**int *src1*, uint *src2***)**	**MPYXU, ACC, T, mem or reg**	The T register is loaded with src1. Src2 is referenced by memory or loaded into a register. The result is in ACC.
long __rpt_**subcu(**long *dst*, int *src1*, int rpt_*cnt***)**	**SUBCU, ACC, mem/reg**	.Rpt_cnt is used as the operand for the RPT instruction. Src2 is referenced from memory or loaded into a register and used as an operand to the SUBCU instruction. The result is in ACC.
long __**sat(**long *src***)**	**SAT, ACC**	Load ACC with 32-bit src. The result is in ACC.
long __**satlow16(**long *src***)**	**SETC OVM** **MOV T, #0xFFFF** **CLR SXM; if necessary** **ADD ACC, T<< 15** **SUB ACC, T 15** **SUB ACC, T 15** **ADD ACC, T 15** **CLRC OVM**	Saturate a 32-bit value to 16-bits low. Load ACC with src. Load T register with #0xFFFF. The result is in ACC.

Table 6–3. TMS320C27xx C Compiler Intrinsics (Continued)

C Compiler Intrinsic	Assembly Instruction	Description
long **__sathi16(**long *src*, int *mask***)**	**SETC OVM** **ADD ACC, mem/reg << 16** **SUB ACC, mem/reg << 16** **SUB ACC, mem/reg << 16** **ADD ACC, mem/reg << 16** **CLRC OVM** **SFR ACC, rshift**	Saturate a 32-bit value to 16-bits high. Load ACC with src. Mask value is either referenced from memory or loaded into register. The result is in ACC. Note: The result can be right shifted and stored into an int. For example: ivar = __sathi16(lvar, mask) >> 6;
long **__sat32(**long *src*, long *mask***)**	**SETC OVM** **ADDL ACC, mem/P** **SUBL ACC, mem/P** **SUBL ACC, mem/P** **ADDL ACC, mem/P** **CLRC OVM**	Saturate a 32-bit value to a 32-bit mask. Load ACC with src. Mask value is either referenced from memory or loaded into the P register. The result is in ACC.

6.5 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 effect on the C environment and can be easily implemented with asm statements.

6.5.1 General Points About Interrupts

An interrupt routine can perform any task performed by any other function, including accessing global variables, allocating local variables, and calling other functions.

When you write interrupt routines, keep the following points in mind:

- ❏ An interrupt handling routine cannot have arguments. If any are declared, they are ignored.
- ❏ An interrupt handling routine can be called by normal C code, but it is inefficient to do this because all the registers are saved.
- ❏ An interrupt handling 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_int00, which is the system reset interrupt. When you enter this routine, you cannot assume that the runtime stack is set up; therefore, you *cannot allocate local variables*, and you *cannot save any information on the runtime stack*.
- ❏ To associate an interrupt routine with an interrupt, the address of the interrupt function must be placed in the appropriate interrupt vector. You can use the assembler and linker to do this by creating a simple table of interrupt addresses using the .sect assembler directive.
- ❏ In assembly language, remember to precede the symbol name with an underscore. For example, refer to c_int00 as _c_int00.

6.5.2 Using C Interrupt Routines

If a C interrupt routine does not call any other functions, only those registers that the interrupt handler uses are saved and restored. However, if a C interrupt routine *does* call other functions, these functions can modify unknown registers that the interrupt handler does not use. For this reason, the compiler saves all the save-on-call registers if any other functions are called.

A C interrupt routine is like any other C function in that it can have local variables and register variables; however, it should be declared with no arguments and should return void. Interrupt handling functions should not be called directly.

Interrupts can be handled *directly* with C functions by using the interrupt pragma or the interrupt keyword. For information about the interrupt pragma, see section 5.6.3 on page 5-13. For information about the interrupt keyword, see section 5.5, *The interrupt Keyword*, on page 5-9.

6.6 Integer Expression Analysis

This section describes some special considerations to keep in mind when evaluating integer expressions.

6.6.1 Integer Operations Evaluated With RTS Calls

The 'C27xx does not directly support some C integer operations. Evaluating these operations is done with calls to run-time-support routines. These routines are hard-coded in assembly language. They are members of the object and source run-time-support libraries (rts.lib and rts.src) in the toolset.

The conventions for calling these routines are modeled on the standard C calling conventions.

Operation Type	Operations Evaluated With Run-Time-Support Calls
16-bit int	Divide (signed)
	Modulus
32-bit long	Divide
	Modulus
	Multiply
	Shift left
	Shift right

6.6.2 C Code Access to the Upper 16 Bits of 16-Bit Multiply

The following methods provide access to the upper 16 bits of a 16-bit multiply in C language:

- Signed-results method:

```
int m1, m2;
int result;

result = ((long) m1 * (long) m2) >> 16;
```

- Unsigned-results method:

```
unsigned m1, m2;
unsigned result;

result = ((unsigned long) m1 * (unsigned long) m2) >> 16;
```

Note: Danger of Complicated Expressions

The compiler must recognize the structure of the expression for it to return the expected results. Avoid complicated expressions such as the following example:

```
((long)((unsigned)((a*b)+c)<5)*(long)(z*sin(w)>6))>>16
```

6.7 Floating-Point Expression Analysis

The ’C27xx 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 ’C27xx run-time-support library, rts.lib, contains a 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

The conventions for calling these routines are the same as the conventions used to call the integer operation routines. Conversions are unary operations.

6.8 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_int00. The run-time-support source library contains the source to this routine in a module called boot.asm.

The c_int00 function can be called by reset hardware to begin running the system. The function is in the run-time-support library (rts.lib) and must be linked with the C object modules. This occurs by default when you use the –c or –cr option in the linker and include rts.lib 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_int00.

The c_int00 function performs the following tasks in order to initialize the C environment:

1) Reserves space in .bss for the runtime stack and sets up the initial stack pointer

2) Initializes global variables by copying the data from the initialization tables in .cinit to the storage allocated for the variables in .bss. In the RAM auto-initialization model, a loader performs this step before the program runs (it is not performed by the boot routine).

3) Calls the function main to run the C program.

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

6.8.1 Runtime Stack

The runtime stack is allocated in a single contiguous block of memory and grows up from low addresses to higher addresses. The SP points to the next available word in the stack.

The code does not 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.

6.8.2 Automatic Initialization of Variables

Some global variables must have initial values assigned to them before a C program starts running. The process of retrieving these variables' data and initializing the variables with the data is called autoinitialization.

The compiler builds tables in a special section called .cinit that contains data for initializing global and static variables. Each compiled module contains these initialization tables. The linker combines them into a single table (a single .cinit section). The boot routine or a loader uses this table to initialize all the system variables.

Note: Initializing Variables

In standard C, global and static variables that are not explicitly initialized are set to 0 before program execution. The 'C27xx C compiler does not perform any preinitialization of uninitialized variables. Explicitly initialize any variable that must have an initial value of 0.

The easiest method is to have a loader clear the .bss section before the program starts running. Another method is to set a fill value of 0 in the linker control map for the .bss section.

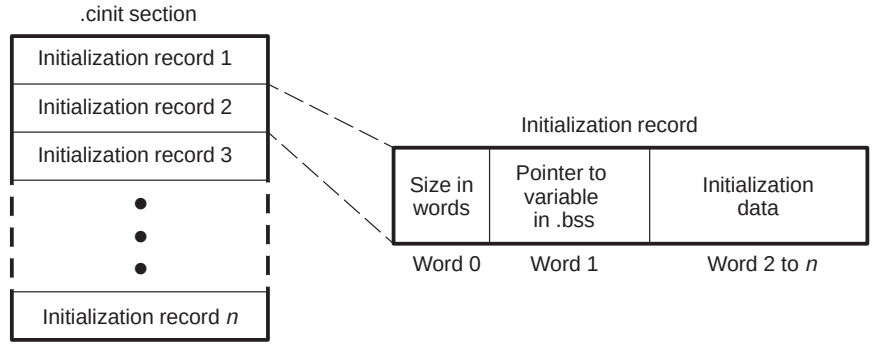
You cannot use these methods with code that is burned into ROM.

Global variables are either autoinitialized at run time or at load time (see sections 6.8.4, *Autoinitialization of Variables at Run Time*, on page 6-32 and 6.8.5, *Autoinitialization of Variables at Load Time*, on page 6-33).

6.8.3 Initialization Tables

The tables in the .cinit section consist of variable-size initialization records. Figure 6–2 shows the format of the .cinit section and the initialization records.

Figure 6–2. Format of Initialization Records in the .cinit Section



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 ".text"

   .sect ".cinit"
   .align 1
   .field   IR_1,16
   .field   _a+0,16
   .field   1,16      ; _a[0] @ 0
   .field   2,16      ; _a[1] @ 16
   .field   3,16      ; _a[2] @ 32
   .field   4,16      ; _a[3] @ 48
   .field   5,16      ; _a[4] @ 64
IR_1: .set  5

   .sect ".text"
```

The .cinit section contains only initialization tables in this format. If you interface assembly language modules to your C programs, 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.

Variables qualified as const are initialized differently; see section 5.7, *Initializing Static and Global Variables*, on page 5-15.

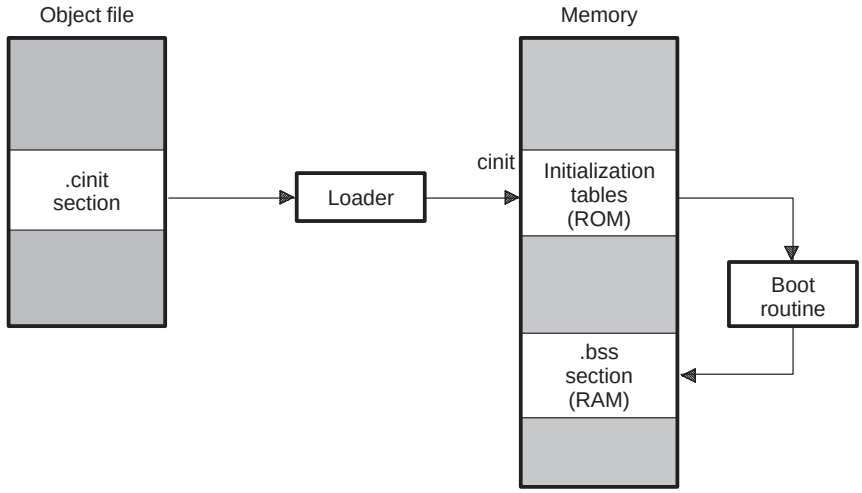
6.8.4 Autoinitialization of Variables at Run Time

Autoinitializing variables at run time is the default method of autoinitialization. To use this method, invoke the linker with the –c option.

Using this method, the .cinit section is loaded into memory along with all the other initialized sections. The linker defines a special symbol called cinit that points to the beginning of the initialization tables in memory. When the program begins running, the C boot routine copies data from the tables (pointed to by .cinit) into the specified variables in the .bss section. This allows initialization data to be stored in ROM and copied to RAM each time the program starts.

Figure 6–3 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into ROM.

Figure 6–3. Autoinitialization at Run Time



6.8.5 Autoinitialization of Variables at Load Time

Autoinitialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the linker with the –cr option.

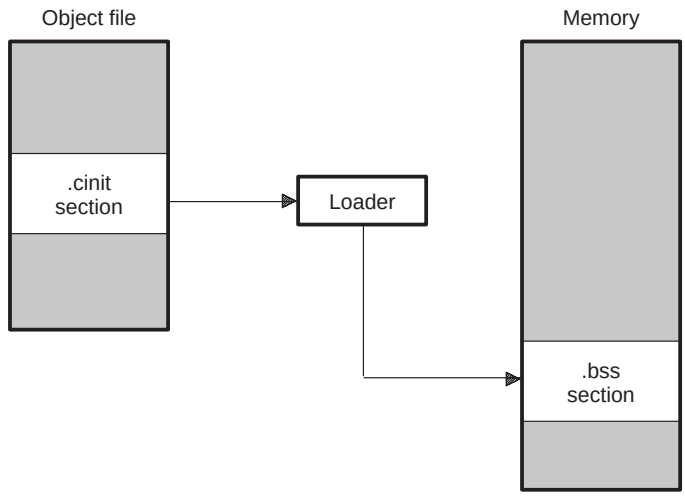
When you use the –cr linker option, the linker sets the STYP_COPY bit in the .cinit section's header. This tells the loader not to load the .cinit section into memory. (The .cinit section occupies no space in the memory map.) The linker also sets the cinit symbol to –1 (normally, cinit points to the beginning of the initialization tables). This indicates to the boot routine that the initialization tables are not present in memory; accordingly, no runtime initialization is performed at boot time.

A loader (which is not part of the compiler package) must be able to perform the following tasks to use autoinitialization at load time:

- Detect the presence of the .cinit section in the object file
- Determine that STYP_COPY is set in the .cinit section header, so that it knows not to copy the .cinit section into memory
- Understand the format of the initialization tables

Figure 6–4 illustrates the autoinitialization of variables at load time.

Figure 6–4. Autoinitialization at Load Time



Chapter 7

Run-Time-Support Functions

Some of the tasks that a C program performs (such as memory allocation, string conversion and string searches) are not part of the C language. The run-time-support functions, which are included in the C compiler, are standard ANSI functions that perform these tasks.

The run-time-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 signals) are provided.

If you use any of the run-time-support functions, be sure to build the appropriate library, according to the device, using the library-build utility (see Chapter 8, *Library-Build Utility*); include that library when you link your C program.

Topic **Page**

7.1 Libraries

The following libraries are included with the TMS320C27xx C compiler:

- *rts.lib* contains the ANSI run-time-support object library
- *rts.src* contains the source for the ANSI run-time-support routines.

The object library includes the standard C run-time-support functions described in this chapter, the floating-point routines, and the system startup routine, _c_int00. the object library is built from the C and assembly source contained in rts.src.

7.1.1 Linking Code With the Object Library

When you link your program, you must specify the object library as one of the linker input files so that references to the I/O and run-time-support functions can be resolved.

You should specify libraries *last* on the linker command line because the linker searches a library for unresolved references when it encounters the library on the command line. You can also use the –x linker option to force repeated searches of each library until the linker can resolve no more references.

When a library is linked, the linker includes only those library members required to resolve undefined references. For more information about linking, see the *TMS320C27xx Assembly Language Tools User's Guide*.

7.1.2 Modifying a Library Function

You can inspect or modify library functions by using the archiver to extract the appropriate source file or files from the source libraries. For example, the following command extracts two source files:

```
ar27 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 reinstall the new object file or files into the library:

```
cl27 –options atoi.c strcpy.c            ;recompile
ar27 r rts.lib  atoi.obj strcpy.obj   ;rebuild library
```

You can also build a new library this way, rather than rebuilding into rts.lib. For more information about the archiver, see the *TMS320C27xx Assembly Language Tools User's Guide*.

7.1.3 Building a Library With Different Options

You can create a new library from rts.src by using the library-build utility mk27. For example, use this command to build an optimized run-time-support library:

```
mk27 –-u –o2 –x rts.src –l rts.lib
```

The –-u option tells the mk27 utility to use the header files in the current directory, rather than extracting them from the source archive. The new library is compatible with any code compiled for the ’C27xx. The use of the (–o2) and inline function expansion (–x) options does not affect compatibility with code compiled without these options. For more information on the library-build utility, see Chapter 8.

7.2 The C I/O Functions

The C I/O functions make it possible to access the host's operating system to perform I/O (using the debugger). For example, printf statements executed in a program appear in the debugger command window. When used in conjunction with the debugging tools, the capability to perform I/O on the host gives you more options when debugging and testing code.

To use the I/O functions, include the header file stdio.h for each module that references a C I/O function.

For example, given the following program in a file named main.c:

```
#include <stdio.h>

main()
{
 FILE *fid;

 fid = fopen("myfile","w");
 fprintf(fid,"Hello, world\n");
 fclose(fid);

 printf("Hello again, world\n");
}
```

Issuing the following shell command compiles, links, and creates the file main.out:

```
cl27 main.c –z –heap 400 –l rts.lib –o main.out
```

Executing main.out under the debugger on a PC host accomplishes the following:

1) Opens the file *myfile* in the directory where the debugger was invoked
2) Prints the string *Hello, world* into that file
3) Closes the file
4) Prints the string *Hello again, world* in the debugger command window

With properly written device drivers, the library also offers facilities to perform I/O on a user-specified device.

If there is not enough space on the heap for a C I/O buffer, buffered operations on the file will fail. If a call to printf() mysteriously fails, this may be the reason. Check the size of the heap. To set the heap size, use the –heap option when linking (see page 4-4).

7.2.1 Overview of Low-Level I/O Implementation

The code that implements I/O is logically divided into layers: high level, low level, and device level.

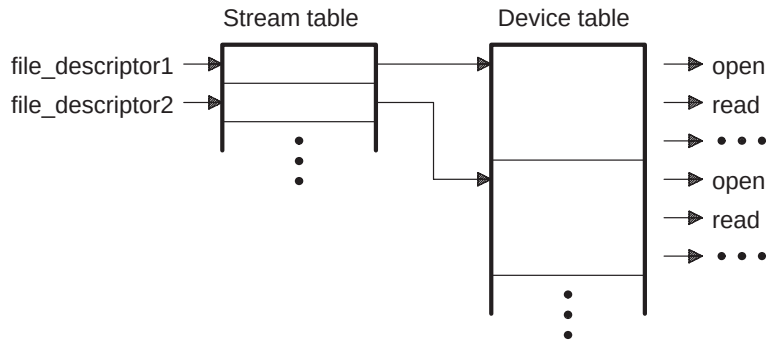
The high-level functions are the standard C library of stream I/O routines (printf, scanf, fopen, getchar, and so on). These routines map an I/O request to one or more of the I/O commands that are handled by the low-level routines.

The low-level routines are comprised of basic I/O functions: OPEN, READ, WRITE, CLOSE, LSEEK, RENAME,and UNLINK. These low-level routines provide the interface between the high-level functions and the device-level drivers that actually perform the I/O command on the specified device.

The low-level functions also define and maintain a stream table that associates a file descriptor with a device. The stream table interacts with the device table to ensure that an I/O command performed on a stream executes the correct device-level routine.

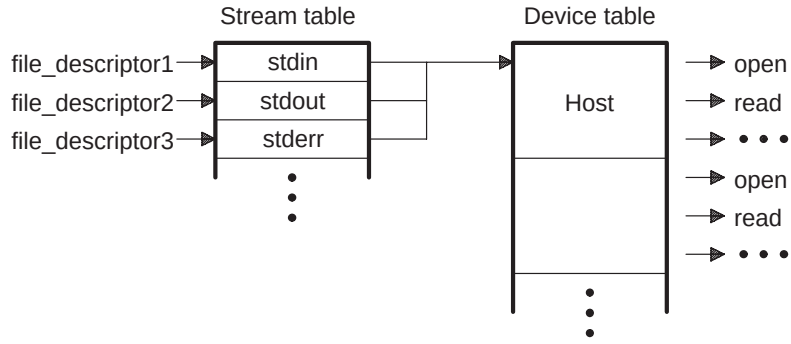
The data structures interact as shown in Figure 7–1.

Figure 7–1. Interaction of Data Structures in I/O Functions



The first three streams in the stream table are predefined to be stdin, stdout, and stderr and they point to the host device and associated device drivers.

Figure 7–2. The First Three Streams in the Stream Table



At the next level are the user-definable device-level drivers. They map directly to the low-level I/O functions. The run-time-support library includes the device drivers necessary to perform I/O on the host on which the debugger is running.

The specifications for writing device-level routines to interface with the low-level routines follow. Each function must set up and maintain its own data structures as needed. Some function definitions perform no action and should just return.

CLOSE — *Close File or Device For I/O*

Syntax

int CLOSE(int *file_descriptor***);**

Description

The CLOSE function closes the device or file associated with *file_descriptor*.

The *file_descriptor* is the stream number assigned by the low-level routines that is associated with the opened device or file.

Return Value

The return value is one of the following:

0 if successful
–1 if fails

LSEEK — *Set File Position Indicator*

Syntax

long LSEEK(int *file_descriptor***, long** *offset***, int** *origin***);**

Description

The LSEEK function sets the file position indicator for the given file to *origin* + *offset*. The file position indicator measures the position in characters from the beginning of the file.

- The *file_descriptor* is the stream number assigned by the low-level routines that the device-level driver must associate with the opened file or device.
- The *offset* indicates the relative offset from the *origin* in characters.
- The *origin* is used to indicate which of the base locations the *offset* is measured from. The *origin* must be a value returned by one of the following macros:

  SEEK_SET (0x0000) Beginning of file
  SEEK_CUR (0x0001) Current value of the file position indicator
  SEEK_END (0x0002) End of file

Return Value

The return function is one of the following:

new value of the file-position indicator if successful
EOF if fails

OPEN — *Open File or Device For I/O*

Syntax

int OPEN(char ****path*, unsigned** *flags***, int** *mode***);**

Description

The OPEN function opens the device or file specified by *path* and prepares it for I/O.

- The *path* is the filename of the file to be opened, including path information.
- The *flags* are attributes that specify how the device or file is manipulated. The flags are specified using the following symbols:

```
O_RDONLY (0x0000) /* open for reading */
O_WRONLY (0x0001) /* open for writing */
O_RDWR   (0x0002) /* open for read & write */
O_APPEND (0x0008) /* append on each write */
O_CREAT  (0x0100) /* open with file create */
O_TRUNC  (0x0200) /* open with truncation */
O_BINARY (0x8000) /* open in binary mode */
```

  These parameters can be ignored in some cases, depending on how data is interpreted by the device. Note, however, that the high-level I/O calls look at how the file was opened in an fopen statement and prevent certain actions, depending on the open attributes.

- The *mode* is required but ignored.

Return Value

The function returns one of the following values:

#	stream number assigned by the low-level routines that the device-level driver associates with the opened file or device if successful
< 0	if fails

READ — *Read Characters From Buffer*

Syntax

int READ(int *file_descriptor*, **char** **buffer*, **unsigned** *count*);

Description

The READ function reads the number of characters specified by *count* to the *buffer* from the device or file associated with *file_descriptor*.

- The *file_descriptor* is the stream number assigned by the low-level routines that is associated with the opened file or device.
- The *buffer* is the location of the buffer where the read characters are placed.
- The *count* is the number of characters to read from the device or file.

Return Value

The function returns one of the following values:

0 if EOF was encountered before the read was complete
\# number of characters read in every other instance
–1 if fails

RENAME — *Rename File*

Syntax

int RENAME(char ***_old_name_, char ***_new_name_**);**

Description

The RENAME function changes the name of a file.

- The *old_name* is the current name of the file.
- The *new_name* is the new name for the file.

Return Value

The function returns one of the following values:

0 if successful
Non-0 if fails

UNLINK — *Delete File*

Syntax **int UNLINK(char *_path_);**

Description The UNLINK function deletes the file specified by *path*.

The *path* is the filename of the file to be opened, including path information.

Return Value The function returns one of the following values:

0 if successful
–1 if fails

WRITE — *Write Characters to Buffer*

Syntax **int WRITE(int** *file_descriptor*, **char *** *buffer*, **unsigned** *count*);

Description The WRITE function writes the number of characters specified by *count* from the *buffer* to the device or file associated with *file_descriptor*.

- The *file_descriptor* is the stream number assigned by the low-level routines that is associated with the opened file or device.
- The *buffer* is the location of the buffer where the write characters are placed.
- The *count* is the number of characters to write to the device or file.

Return Value The function returns one of the following values:

number of characters written if successful
–1 if fails

7.2.2 Adding a Device for C I/O

The low-level functions provide facilities that allow you to add and use a device for I/O at run time. The procedure for using these facilities is:

1) Define the device-level functions as described in section 7.2.1, *Overview of Low-Level I/O Implementation,* on page 7-5.

> **Note: Use Unique Function Names**
>
> The function names open, close, read, and so on, are used by the low-level routines. Use other names for the device-level functions that you write.

2) Use the low-level function add_device() to add your device to the device_table. The device table is a statically defined array that supports *n* devices, where *n* is defined by the macro _NDEVICE found in stdio.h. The structure representing a device is also defined in stdio.h and is composed of the following fields:

name	String for device name
flags	Specifies whether the device supports multiple streams or not
function pointers	Pointers to the device-level functions: ❑ CLOSE ❑ LSEEK ❑ OPEN ❑ READ ❑ RENAME ❑ WRITE ❑ UNLINK

The first entry in the device table is predefined to be the host device on which the debugger is running. The low-level routine add_device() finds the first empty position in the device table and initializes the device fields with the passed-in arguments. For a complete description, see the add_device function on page 7-32.

3) Once the device is added, call fopen() to open a stream and associate it with that device. Use *devicename*:*filename* as the first argument to fopen().

The following program illustrates adding and using a device for C I/O:

```
#include <stdio.h>

/*************************************************************************/
/* Declarations of the user-defined device drivers                       */
/*************************************************************************/
extern int my_open(char *path, unsigned flags, int fno);
extern int my_close(int fno);
extern int my_read(int fno, char *buffer, unsigned count);
extern int my_write(int fno, char *buffer, unsigned count);
extern int my_lseek(int fno, long offset, int origin);
extern int my_unlink(char *path);
extern int my_rename(char *old_name, char *new_name);

main()
{
   FILE *fid;

   add_device("mydevice", _MSA, my_open, my_close, my_read, my_write, my_lseek,

                                my_unlink, my_rename);

   fid = fopen("mydevice:test","w");

   fprintf(fid,"Hello, world\n");

   fclose(fid);
}
```

7.3 Header Files

Each run-time-support function is declared in a *header file*. Each header file declares the following:

- A set of related functions (or macros)
- Any types that you need to use the functions
- Any macros that you need to use the functions

These are the header files that declare the ANSI C standard run-time-support functions:

```
assert.h      float.h       stdarg.h      string.h
ctype.h       limits.h      stddef.h      time.h
errno.h       math.h        stdio.h
file.h        setjmp.h      stdlib.h
```

In order to use a run-time-support function, you must first use the #include preprocessor directive to include the header file that declares the function. For example, the isdigit function is declared by the ctype.h header. Before you can use the isdigit function, you must first include ctype.h:

```
#include <ctype.h>
   .
   .
   .
   val = isdigit(num);
```

You can include headers in any order. You must, however, include a header before you reference any of the functions or objects that it declares.

Sections 7.3.1, *Diagnostic Messages (assert.h)*, on page 7-14 through 7.3.13, *Time Functions (time.h)*, on page 7-21 describe the header files that are included with the 'C27xx C compiler. Section 7.4, *Summary of run-time-Support Functions and Macros*, on page 7-23 lists the functions that these headers declare.

7.3.1 Diagnostic Messages (assert.h)

The assert.h header defines the assert macro, which inserts diagnostic failure messages into programs at run time. The assert macro tests a 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 (using the abort function).

The assert.h header refers to another macro named NDEBUG (assert.h does not define NDEBUG). If you have defined NDEBUG as a macro name when you include assert.h, assert is turned off and does nothing. If NDEBUG is *not* defined, assert is enabled.

The assert.h header refers to another macro named NASSERT (assert.h does not define NASSERT). If you have defined NASSERT as a macro name when you include assert.h, assert acts like _nassert. The _nassert intrinsic generates no code and tells the optimizer that the expression declared with assert is true. This gives a hint to the optimizer as to what optimizations might be valid. If NASSERT is *not* defined, assert is enabled normally.

The assert function is listed in Table 7–3(a) on page 7-23.

7.3.2 Character-Typing and Conversion (ctype.h)

The ctype.h header declares functions that test (type) and convert characters.

The character-typing functions test a character to determine whether it is a letter, a printing character, a hexadecimal digit, etc. These functions return a value of *true* (a nonzero value) or *false* (0). Character-typing functions have names in the form is*xxx* (for example, *isdigit*).

The character-conversion functions convert characters to lowercase, uppercase, or ASCII, and return the converted character. Character-conversion functions have names in the form to*xxx* (for example, *toupper*).

The ctype.h header also contains macro definitions that perform these same operations. The macros run faster than the corresponding functions. Use the function version if an argument is passed that has side effects. 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*).

The character typing and conversion functions are listed in Table 7–3(b) on page 7-23.

7.3.3 Error Reporting (errno.h)

The errno.h header declares the errno variable. The errno variable indicates errors in math functions. 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)
- ENOENT for path errors (path does not exist)
- EFPOS for seek errors (file position error)

C code that calls a math function can read the value of errno to check for error conditions. The errno variable is declared in errno.h and defined in errno.c.

7.3.4 Low-Level Input/Output Functions (file.h)

The file.h header declares the low-level I/O functions used to implement input and output operations.

How to implement I/O for the ’C27xx is described in section 7.2, *The C I/O Functions*, on page 7-4.

7.3.5 Limits (float.h and limits.h)

The float.h and limits.h headers define macros that expand to useful limits and parameters of the TMS320C27xx’s numeric representations. Table 7–1 and Table 7–2 list these macros and their limits.

Table 7–1. Macros That Supply Integer Type Range Limits (limits.h)

Macro	Value	Description
CHAR_BIT	8	Number of bits in type char
SCHAR_MIN	–128	Minimum value for a signed char
SCHAR_MAX	127	Maximum value for a signed char
UCHAR_MAX	255	Maximum value for an unsigned char
CHAR_MIN	SCHAR_MIN	Minimum value for a char
CHAR_MAX	SCHAR_MAX	Maximum value for a char
SHRT_MIN	–32 768	Minimum value for a short int
SHRT_MAX	32 767	Maximum value for a short int
USHRT_MAX	65 535	Maximum value for an unsigned short int
INT_MIN	(–INT_MAX – 1)	Minimum value for an int
INT_MAX	2 147 483 647	Maximum value for an int
UINT_MAX	4 294 967 295	Maximum value for an unsigned int
LONG_MIN	(–LONG_MAX – 1)	Minimum value for a long int
LONG_MAX	549 755 813 887	Maximum value for a long int
ULONG_MAX	1 099 511 627 775	Maximum value for an unsigned long int

Note: Negative values in this table are defined as expressions in the actual header file so that their type is correct.

Table 7–2. Macros That Supply Floating-Point Range Limits (float.h)

Macro	Value	Description
FLT_RADIX	2	Base or radix of exponent representation
FLT_ROUNDS	1	Rounding mode for floating-point addition
FLT_DIG DBL_DIG LDBL_DIG	6 15 15	Number of decimal digits of precision for a float, double, or long double
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	24 53 53	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 –1021 –1021	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 1024 1024	Maximum negative integer such that FLT_RADIX raised to that power minus 1 is a representable finite float, double, or long double
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	1.19209290e–07 2.22044605e–16 2.22044605e–16	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–38 2.22507386e–308 2.22507386e–308	Minimum positive float, double, or long double
FLT_MAX DBL_MAX LDBL_MAX	3.40282347e+38 1.79769313e+308 1.79769313e+308	Maximum float, double, or long double
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	–37 –307 –307	Minimum negative integers 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	38 308 308	Maximum positive integers such that 10 raised to that power is in the range of representable finite floats, doubles, or long doubles

Legend: FLT_ applies to type float.
DBL_ applies to type double.
LDBL_ applies to type long double.

Note: The precision of some of the values in this table has been reduced for readability. Refer to the float.h header file supplied with the compiler for the full precision carried by the processor.

7.3.6 Floating-Point Math (math.h)

The math.h header declares several trigonometric, exponential, and hyperbolic math functions. These functions are listed in Table 7–3(c) on page 7-24. The math functions expect arguments of type double and return values of type double. Except where indicated, all trigonometric functions use angles expressed in radians.

The math.h header also defines one macro named HUGE_VAL. The math functions use this macro to represent out-of-range values. When a function produces a floating-point return value that is too large to represent, it returns HUGE_VAL instead.

7.3.7 Nonlocal Jumps (setjmp.h)

The setjmp.h header defines a type and a macro and declares a function for bypassing the normal function call and return discipline. These include:

- *jmp_buf*, an array type suitable for holding the information needed to restore a calling environment
- *setjmp*, a macro that saves its calling environment in its jmp_buf argument for later use by the longjmp function
- *longjmp*, a function that uses its jmp_buf argument to restore the program environment. The nonlocal jmp macro and function are listed in Table 7–3(d) on page 7-25.

7.3.8 Variable Arguments (stdarg.h)

Some functions can have a variable number of arguments whose types can differ. Such functions are called *variable-argument functions*. The stdarg.h header declares macros and a type that help you to use variable-argument functions.

- The macros are *va_start*, *va_arg*, and *va_end*. These macros are used when the number and type of arguments can vary each time a function is called.
- The type *va_list* is a pointer type that can hold information for va_start, va_end, and va_arg.

A variable-argument function can use the macros declared by stdarg.h to step through its argument list at run time when the function knows the number and types of arguments actually passed to it. You must ensure that a call to a variable-argument function has visibility to a prototype for the function in order for the arguments to be handled correctly. The variable argument functions are listed in Table 7–3(e) page 7-25.

7.3.9 Standard Definitions (stddef.h)

The stddef.h header defines types and macros. The types are:

- *ptrdiff_t*, a signed integer type that is the data type resulting from the subtraction of two pointers
- *size_t*, an unsigned integer type that is the data type of the *sizeof* operator

The macros are:

- *NULL*, a macro that expands to a null pointer constant(0)
- *offsetof(type, identifier)*, a macro that expands to an integer that has type size_t. The result is the value of an offset in bytes to a structure member (*identifier*) from the beginning of its structure (*type*).

These types and macros are used by several of the run-time-support functions.

7.3.10 Input/Output Functions (stdio.h)

The stdio.h header defines types and macros and declares functions. The types are:

- *size_t*, an unsigned integer type that is the data type of the *sizeof* operator. Originally defined in stddef.h.
- *fpos_t*, an unsigned integer type that can uniquely specify every position within a file.
- *FILE*, a structure type to record all the information necessary to control a stream.

The macros are:

- *NULL*, a macro that expands to a null pointer constant(0). Originally defined in stddef.h. It is not redefined if it was already defined.
- *BUFSIZ*, a macro that expands to the size of the buffer that setbuf() uses.
- *EOF*, the end-of-file marker.
- *FOPEN_MAX*, a macro that expands to the largest number of files that can be open at one time.
- *FILENAME_MAX*, a macro that expands to the length of the longest file name in characters.

- *L_tmpnam*, a macro that expands to the longest filename string that tmpnam() can generate.
- *SEEK_CUR*, *SEEK_SET*, and *SEEK_END*, macros that expand to indicate the position (current, start-of-file, or end-of-file, respectively) in a file.
- *TMP_MAX*, a macro that expands to the maximum number of unique filenames that tmpnam() can generate.
- *stderr*, *stdin*, *stdout*, pointers to the standard error, input, and output files, respectively.

The input/output functions are listed in Table 7–3(f) on page 7-25.

7.3.11 General Utilities (stdlib.h)

The stdlib.h header defines a macro and types and declares functions. The macro is named RAND_MAX, and it returns the largest value returned by the rand() function. The types are:

- *div_t*, a structure type that is the type of the value returned by the div function
- *ldiv_t*, a structure type that is the type of the value returned by the ldiv function

The functions are:

- String conversion functions that convert strings to numeric representations
- Searching and sorting functions that search and sort arrays
- Sequence-generation functions that generate a pseudo-random sequence and choose a starting point for a sequence
- Program-exit functions that terminate your program normally or abnormally
- Integer-arithmetic that is not provided as a standard part of the C language

The general utility functions are listed in Table 7–3(g) page 7-28.

7.3.12 String Functions (string.h)

The string.h header declares standard functions that perform the following tasks with character arrays (strings):

- Move or copy entire strings or portions of strings
- Concatenate strings
- Compare strings
- Search strings for characters or other strings
- Find the length of a string

In C, all character strings are terminated with a 0 (null) character. The string functions named str*xxx* all operate according to this convention. Additional functions that are also declared in string.h perform corresponding operations on arbitrary sequences of bytes (data objects), where a 0 value does not terminate the object. These functions are named mem*xxx*.

When you use functions that move or copy strings, be sure that the destination is large enough to contain the result. The string functions are listed in Table 7–3(h) page 7-29.

7.3.13 Time Functions (time.h)

The time.h header defines one macro and several types, and declares functions that manipulate dates and times. Times are represented in the following ways:

- As an arithmetic value of type time_t. When expressed in this way, a time is represented as a number of seconds since 12:00 AM January 1, 1900. The time_t type is a synonym for the type unsigned long.

- As a structure of type struct_tm. This structure contains members for expressing time as a combination of years, months, days, hours, minutes, and seconds. A time represented like this is 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      */
```

A time, whether represented as a time_t or a struct_tm, can be expressed from different points of reference:

- Calendar time represents the current Gregorian date and time.
- Local time is the calendar time expressed for a specific time zone.

The time functions and macros are listed in Table 7–3(i) on page 7-30.

You can adjust local time for local or seasonal variations. Obviously, local time depends on the time zone. The time.h header defines a structure type called tmzone and a variable of this type called _tz. You can change the time zone by modifying this structure, either at run time or by editing tmzone.c and changing the initialization. The default time zone is CST (Central Standard Time), U.S.A.

The basis for all the time.h functions are the system functions of clock and time. Time provides the current time (in time_t format), and clock provides the system time (in arbitrary units). You can divide the value returned by clock by the macro CLOCKS_PER_SEC to convert it to seconds. Since these functions and the CLOCKS_PER_SEC macro are system specific, only stubs are provided in the library. To use the other time functions, you must supply custom versions of these functions.

Note: Writing Your Own Clock Function

The clock function is host-system specific, so you must write your own clock function or use the clock function supplied with the simulator. You must also define the CLOCKS_PER_SEC macro according to the units of measure 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.

7.4 Summary of Run-Time-Support Functions and Macros

Table 7–3 summarizes the run-time-support header files (in alphabetical order) provided with the TMS320C27xx ANSI C compiler. Most of the functions described are per the ANSI standard and behave exactly as described in the standard.

The functions and macros listed in Table 7–3 are described in detail in section 7.5, *Description of Run-Time-Support Functions and Macros*, on page 7-31. For a complete description of a function or macro, see the indicated page.

Table 7–3. Summary of Run-Time-Support Functions and Macros

(a) Error message macro (assert.h)

Macro	Description	Page
void **assert**(int expr);	Inserts diagnostic messages into programs; expands inline if –x is used	7-34

(b) Character typing and conversion functions (ctype.h)

Function	Description	Page
int **isalnum**(int c);	Tests c to see if it's an alphanumeric-ASCII character	7-50
int **isalpha**(int c);	Tests c to see if it's an alphabetic-ASCII character	7-50
int **isascii**(int c);	Tests c to see if it's an ASCII character	7-50
int **iscntrl**(int c);	Tests c to see if it's a control character	7-50
int **isdigit**(int c);	Tests c to see if it's a numeric character	7-50
int **isgraph**(int c);	Tests c to see if it's any printing character except a space	7-50
int **islower**(int c);	Tests c to see if it's a lowercase alphabetic ASCII charcter	7-50
int **isprint**(int c);	Tests c to see if it's a printable ASCII character (including a space)	7-50
int **ispunct**(int c);	Tests c to see if it's an ASCII punctuation character	7-50
int **isspace**(int c);	Tests c to see if it's an ASCII spacebar, tab (horizontal or vertical), carriage return, formfeed, or newline character	7-50
int **isupper**(int c);	Tests c to see if it's an uppercase ASCII alphabetic character	7-50
int **isxdigit**(int c);	Tests c to see if it's a hexadecimal digit	7-50
char **toascii**(int c);	Masks c into a legal ASCII value	7-78
char **tolower**(int char c);	Converts c to lowercase if it's uppercase	7-78
char **toupper**(int char c);	Converts c to uppercase if it's lowercase	7-78

Note: Functions in ctype.h are expanded inline if the –x option is used.

Table 7–3. Summary of Run-Time-Support Functions and Macros (Continued)

(c) Floating-point math functions (math.h)

Function	Description	Page
double **acos**(double x);	Returns the arc cosine of x	7-31
double **asin**(double x);	Returns the arc sine of x	7-34
double **atan**(double x);	Returns the arc tangent of x	7-35
double **atan2**(double y, double x);	Returns the arc tangent of y/x	7-35
double **ceil**(double x);	Returns the smallest integer greater than or equal to x; expands inline if –x is used	7-38
double **cos**(double x);	Returns the cosine of x	7-39
double **cosh**(double x);	Returns the hyperbolic cosine of x	7-39
double **exp**(double x);	Returns the exponential function of x; expands inline unless –x0 is used	7-42
double **fabs**(double x);	Returns the absolute value of x	7-42
double **floor**(double x);	Returns the largest integer less than or equal to x; expands inline if –x is used	7-44
double **fmod**(double x, double y);	Returns the floating-point remainder of x/y; expands inline if –x is used	7-45
double **frexp**(double value, int *exp);	Breaks the value into a normalized fraction and an integer power of 2	7-47
double **ldexp**(double x, int exp);	Multiplies x by an integer power of 2	7-51
double **log**(double x);	Returns the natural logarithm of x	7-52
double **log10**(double x);	Returns the base-10 logarithm of x	7-52
double **modf**(double value, double *iptr);	Breaks value into a signed integer and a signed fraction	7-57
double **pow**(double x, double y);	Returns x raised to the power y	7-57
double **sin**(double x);	Returns the sine of x	7-63
double **sinh**(double x);	Returns the hyperbolic sine of x	7-64
double **sqrt**(double x);	Returns the nonnegative square root of x	7-64
double **tan**(double x);	Returns the tangent of x	7-76
double **tanh**(double x);	Returns the hyperbolic tangent of x	7-77

Table 7–3. Summary of Run-Time-Support Functions and Macros (Continued)

(d) Nonlocal jumps macro and function (setjmp.h)

Function or Macro	Description	Page
int **setjmp**(jmp_buf env);	Saves calling environment for use by longjmp; this is a macro	7-62
void **longjmp**(jmp_buf env, int _val);	Uses jmp_buf argument to restore a previously saved environment	7-62

(e) Variable argument macros (stdarg.h)

Macro	Description	Page
type **va_arg**(_ap, type);	Accesses the next argument of type type in a variable-argument list	7-79
void **va_end**(_ap);	Resets the calling mechanism after using va_arg	7-79
void **va_start**(_ap, parmN);	Initializes ap to point to the first operand in the variable-argument list	7-79

(f) C I/O functions (stdio.h)

Function	Description	Page
int **add_device**(char *name, unsigned flags, int (*dopen)(), int (*dclose)(), int (*dread)(), int (*dwrite)(), fpos_t (*dlseek)(), int (*dunlink)(), int (*drename)());	Adds a device record to the device table	7-32
void **clearerr**(FILE *_fp);	Clears the EOF and error indicators for the stream that _fp points to	7-38
int **fclose**(FILE *_fp);	Flushes the stream that _fp points to and closes the file associated with that stream	7-43
int **feof**(FILE *_fp);	Tests the EOF indicator for the stream that _fp points to	7-43
int **ferror**(FILE *_fp);	Tests the error indicator for the stream that _fp points to	7-43
int **fflush**(register FILE *_fp);	Flushes the I/O buffer for the stream that _fp points to	7-43
int **fgetc**(register FILE *_fp);	Reads the next character in the stream that _fp points to.	7-43
int **fgetpos**(FILE *_fp, fpos_t *pos);	Stores the object that pos points to to the current value of the file position indicator for the stream that _fp points to	7-44
char ***fgets**(char *_ptr, register int _size, register FILE *_fp);	Reads the next _size minus 1 characters from the stream that _fp points to into array _ptr	7-44

Table 7–3. Summary of Run-Time-Support Functions and Macros (Continued)
(f) C I/O functions (stdio.h) (continued)

Function	Description	Page
FILE ***fopen**(const char *_fname, const char *_mode);	Opens the file that _fname points to; _mode points to a string describing how to open the file	7-45
int **fprintf**(FILE *_fp, const char *_format, ...);	Writes to the stream that _fp points to	7-45
int **fputc**(int _c, register FILE *_fp);	Writes a single character, _c, to the stream that _fp points to	7-45
int **fputs**(const char *_ptr, register FILE *_fp);	Writes the string pointed to by _ptr to the stream pointed to by _fp	7-46
size_t **fread**(void *_ptr, size_t _size, size_t _count, FILE *_fp);	Reads from the stream pointed to by _fp and stores the input to the array pointed to by _ptr	7-46
FILE ***freopen**(const char *_fname, const char *_mode, register FILE *_fp);	Opens the file that _fname points to using the stream that _fp points to; _mode points to a string describing how to open the file	7-47
int **fscanf**(FILE *_fp, const char *_fmt, ...);	Reads formatted input from the stream that _fp points to	7-47
int **fseek**(register FILE *_fp, long _offset, int _ptrname);	Sets the file position indicator for the stream that _fp points to	7-47
int **fsetpos**(FILE *_fp, const fpos_t *_pos);	Sets the file position indicator for the stream that _fp points to to _pos. The pointer _pos must be a value from fgetpos() on the same stream.	7-48
long **ftell**(FILE *_fp);	Obtains the current value of the file position indicator for the stream that _fp points to	7-48
size_t **fwrite**(const void *_ptr, size_t _size, size_t _count, register FILE *_fp);	Writes a block of data from the memory pointed to by _ptr to the stream that _fp points to	7-48
int **getc**(FILE *_fp);	Reads the next character in the stream that _fp points to	7-48
int **getchar**(void);	A macro that calls fgetc() and supplies stdin as the argument	7-49
char ***gets**(char *_ptr);	Performs the same function as fgets() using stdin as the input stream	7-49
void **perror**(const char *_s);	Maps the error number in _s to a string and prints the error message	7-57
int **printf**(const char *_format, ...);	Performs the same function as fprintf but uses stdout as its output stream	7-58
int **putc**(int _x, FILE *_fp);	A macro that performs like fputc()	7-58
int **putchar**(int _x);	A macro that calls fputc() and uses stdout as the output stream	7-58

Table 7–3. Summary of Run-Time-Support Functions and Macros (Continued)

(f) C I/O functions (stdio.h) (continued)

Function	Description	Page
int **puts**(const char *_ptr);	Writes the string pointed to by _ptr to stdout	7-58
int **remove**(const char *_file);	Causes the file with the name pointed to by _file to be no longer available by that name	7-60
int **rename**(const char *_old_name, const char *_new_name);	Causes the file with the name pointed to by _old_name to be known by the name pointed to by _new_name	7-61
void **rewind**(register FILE *_fp);	Sets the file position indicator for the stream pointed to by _fp to the beginning of the file	7-61
int **scanf**(const char *_fmt, ...);	Performs the same function as fscanf() but reads input from stdin	7-61
void **setbuf**(register FILE *_fp, char *_buf);	Returns no value. setbuf() is a restricted version of setvbuf() and defines and associates a buffer with a stream	7-61
int **setvbuf**(register FILE *_fp, register char *_buf, register int _type, register size_t _size);	Defines and associates a buffer with a stream	7-63
int **sprintf**(char *_string, const char *_format, ...);	Performs the same function as fprintf() but writes to the array that _string points to	7-64
int **sscanf** (const char *_str, const char *_fmt, ...);	Performs the same function as fscanf() but reads from the string that _str points to	7-65
FILE ***tmpfile**(void);	Creates a temporary file	7-77
char ***tmpnam**(char *_s);	Generates a string that is a valid filename (that is, the filename is not already being used)	7-78
int **ungetc**(int _c, register FILE *_fp);	Pushes the character specified by _c back into the input stream pointed to by _fp	7-78
int **vfprintf**(FILE *_fp, const char *_format, va_list _ap);	Performs the same function as fprintf() but replaces the argument list with _ap	7-80
int **vprintf**(const char *_format, va_list _ap);	Performs the same function as printf() but replaces the argument list with _ap	7-80
int **vsprintf**(char *_string, const char *_format, va_list _ap);	Performs the same function as sprintf() but replaces the argument list with _ap	7-80

Table 7–3. Summary of Run-Time-Support Functions and Macros (Continued)

(g) General functions (stdlib.h)

Function	Description	Page
void **abort**(void);	Terminates a program abnormally	7-31
int **abs**(int i);	Returns the absolute value of val; expands inline unless –x0 is used	7-31
int **atexit**(void (*fun)(void));	Registers the function pointed to by fun, called without arguments at program termination	7-35
double **atof**(const char *st);	Converts a string to a floating-point value; expands inline if –x is used	7-36
int **atoi**(register const char *st);	Converts a string to an integer	7-36
long **atol**(register const char *st);	Converts a string to a long integer value; expands inline if –x is used	7-36
void ***bsearch**(register const void *key, register const void *base, size_t nmemb, size_t size, int (*compar)(const void *,const void *));	Searches through an array of nmemb objects for the object that key points to	7-37
void ***calloc**(size_t num, size_t size);	Allocates and clears memory for num objects, each of size bytes	7-37
div_t **div**(register int numer, register int denom);	Divides numer by denom producing a quotient and a remainder	7-41
void **exit**(int status);	Terminates a program normally	7-42
void **free**(void *packet);	Deallocates memory space allocated by malloc, calloc, or realloc	7-46
char ***getenv**(const char *_string)	Returns the environment information for the variable associated with _string	7-49
long **labs**(long i);	Returns the absolute value of i; expands inline unless –x0 is used	7-31
ldiv_t **ldiv**(register long numer, register long denom);	Divides numer by denom	7-41
int **ltoa**(long val, char *buffer);	Converts val to the equivalent string	7-52
void ***malloc**(size_t size);	Allocates memory for an object of size bytes	7-53
void ***memalign**(size_t alignment, size_t size);	Allocates memory for an object of size bytes aligned to an alignment byte boundary	7-53
void **minit**(void);	Resets all the memory previously allocated by malloc, calloc, or realloc	7-56
void **qsort**(void *base, size_t nmemb, size_t size, int (*compar) ());	Sorts an array of nmemb members; base points to the first member of the unsorted array, and size specifies the size of each member	7-59

Table 7–3. Summary of Run-Time-Support Functions and Macros (Continued)

(g) General utilities (stdlib.h) (continued)

Function	Description	Page
int **rand**(void);	Returns a sequence of pseudorandom integers in the range 0 to RAND_MAX	7-59
void ***realloc**(void *packet, size_t size);	Changes the size of an allocated memory space	7-60
void **srand**(unsigned int seed);	Resets the random number generator	7-59
double **strtod**(const char *st, char **endptr);	Converts a string to a floating-point value	7-75
long **strtol**(const char *st, char **endptr, int base);	Converts a string to a long integer	7-75
unsigned long **strtoul**(const char *st, char **endptr**,** int base)**;**	Converts a string to an unsigned long integer	7-75

(h) String functions (string.h)

Function	Description	Page
void ***memchr**(const void *cs, int c, size_t n);	Finds the first occurrence of c in the first n characters of cs; expands inline if –x is used	7-53
int **memcmp**(const void *cs, const void *ct, size_t n);	Compares the first n characters of cs to ct; expands inline if –x is used	7-54
void ***memcpy**(void *s1, const void *s2, register size_t n);	Copies n characters from s1 to s2	7-54
void ***memmove**(void *s1, const void *s2, size_t n);	Moves n characters from s1 to s2	7-54
void ***memset**(void *mem, register int ch, register size_t length);	Copies the value of ch into the first length characters of mem; expands inline if –x is used	7-55
char ***strcat**(char *string1, const char *string2);	Appends string2 to the end of string1	7-66
char ***strchr**(const char *string, int c);	Finds the first occurrence of character c in s; expands inline if –x is used	7-66
int **strcmp**(register const char *string1, register const char *s2);	Compares strings and returns one of the following values: <0 if string1 is less than string2; 0 if string1 is equal to string2; >0 if string1 is greater than string2. Expands inline if –x is used.	7-67
int **strcoll**(const char *string1, const char *string2);	Compares strings and returns one of the following values: <0 if string1 is less than string2; 0 if string1 is equal to string2; >0 if string1 is greater than string2.	7-67
char ***strcpy**(register char *dest, register const char *src);	Copies string src into dest; expands inline if –x is used	7-67
size_t **strcspn**(register const char *string, const char *chs);	Returns the length of the initial segment of string that is made up entirely of characters that are not in chs	7-68

Table 7–3. Summary of Run-Time-Support Functions and Macros (Continued)

(h) String functions (string.h) (continued)

Function	Description	Page
char ***strerror**(int errno);	Maps the error number in errno to an error message string	7-68
size_t **strlen**(const char *string);	Returns the length of a string	7-70
char ***strncat**(char *dest, const char *src, register size_t n);	Appends up to n characters from src to dest	7-70
int **strncmp**(const char *string1, const char *string2, size_t n);	Compares up to n characters in two strings; expands inline if –x is used	7-71
char ***strncpy**(register char *dest, register const char *src, register size_t n);	Copies up to n characters from src to dest; expands inline if –x is used	7-72
char ***strpbrk**(const char *string, const char *chs);	Locates the first occurrence in string of *any* character from chs	7-73
char ***strrchr**(const char *string, int c);	Finds the last occurrence of character c in string; expands inline if –x is used	7-73
size_t **strspn**(register const char *string, const char *chs);	Returns the length of the initial segment of string, which is entirely made up of characters from chs	7-74
char ***strstr**(register const char *string1, const char *string2);	Finds the first occurrence of string2 in string1	7-74
char ***strtok**(char *str1, const char *str2);	Breaks str1 into a series of tokens, each delimited by a character from str2	7-76
size_t **strxfrm**(register char *to, register const char *from, register size_t n);	Transforms n characters from from, to to	7-76

(i) Time-support functions (time.h)

Function	Description	Page
char ***asctime**(const struct tm *timeptr);	Converts a time to a string	7-33
clock_t **clock**(void);	Determines the processor time used	7-38
char ***ctime**(const time_t *timer);	Converts calendar time to local time	7-39
double **difftime**(time_t time1, time_t time0);	Returns the difference between two calendar times	7-41
struct tm ***gmtime**(const time_t *timer);	Converts local time to Greenwich Mean Time	7-49
struct tm ***localtime**(const time_t *timer);	Converts time_t value to broken down time	7-51
time_t **mktime**(register struct tm *tptr);	Converts broken down time to a time_t value	7-55
size_t **strftime**(char *out, size_t maxsize, const char *format, const struct tm *time);	Formats a time into a character string	7-69
time_t **time**(time_t *timer);	Returns the current calendar time	7-77

7.5 Description of Run-Time-Support Functions and Macros

abort — *Abort*

Syntax

#include <stdlib.h>

void abort(void**);**

Defined in

exit.c in rts.src

Description

The abort function terminates the program.

Example

```
void abort(void)
{
   exit(EXIT_FAILURE);
}
```

See the exit function on page 7-42.

abs/labs — *Absolute Value*

Syntax

#include <stdlib.h>

int abs(int i**);**
long labs(long i**);**

Defined in

abs.c in 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 i.
- The labs function returns the absolute value of a long i.

acos — *Arc Cosine*

Syntax

#include <math.h>

double acos(double x**);**

Defined in

acos.c in rts.src

Description

The acos function returns the arc cosine of a floating-point argument x, which must be in the range [–1,1]. The return value is an angle in the range [0,π] radians.

Example

```
double realval, radians;

return (realval = 1.0;
radians = acos(realval);
return (radians);  /* acos return π/2 */
```

add_device — *Add Device to Device Table*

Syntax

```
#include <stdio.h>

int add_device(char *name,
               unsigned flags,
               int (*dopen)(),
               int (*dclose)(),
               int (*dread)(),
               int (*dwrite)(),
               fpos_t (*dlseek)(),
               int (*dunlink)(),
               int (*drename)());
```

Defined in

lowlev.c in rts.src

Description

The add_device function adds a device record to the device table allowing that device to be used for input/output from C. The first entry in the device table is predefined to be the host device on which the debugger is running. The function add_device() finds the first empty position in the device table and initializes the fields of the structure that represent a device.

To open a stream on a newly-added device use fopen() with a string of the format *devicename*:*filename* as the first argument.

- The *name* is a character string denoting the device name.
- The *flags* are device characteristics. The flags are as follows:

  _SSA Denotes that the device supports only one open stream at a time

  _MSA Denotes that the device supports multiple open streams

  More flags can be added by defining them in stdio.h.

- The dopen, dclose, dread, dwrite, dlseek, dunlink, drename specifiers are function pointers to the device drivers that are called by the low-level functions to perform I/O on the specified device. You must declare these functions with the interface specified in section 7.2.1, *Overview of Low-Level I/O Implementation*, on page 7-5. The device drivers for the host that the TMS320C27xx debugger is run on are included in the C I/O library.

Return Value

The function returns one of the following values:

0 if successful

–1 if fails

Example This example does the following:

- ❏ Adds the device *mydevice* to the device table
- ❏ Opens a file named *test* on that device and associate it with the file **fid*
- ❏ Writes the string *Hello, world* into the file
- ❏ Closes the file

```
#include <stdio.h>

/*************************************************************************/
/* Declarations of the user-defined device drivers                       */
/*************************************************************************/
extern int my_open(char *path, unsigned flags, int fno);
extern int my_close(int fno);
extern int my_read(int fno, char *buffer, unsigned count);
extern int my_write(int fno, char *buffer, unsigned count);
extern int my_lseek(int fno, long offset, int origin);
extern int my_unlink(char *path);
extern int my_rename(char *old_name, char *new_name);

main()
{

   FILE *fid;
   add_device("mydevice", _MSA, my_open, my_close, my_read, my_write, my_lseek,
                                my_unlink, my_rename);

   fid = fopen("mydevice:test","w");

   fprintf(fid,"Hello, world\n");

   fclose(fid);
}
```

asctime — *Convert Internal Time to String*

Syntax #include <time.h>

char *asctime(const struct tm *timeptr**);**

Defined in asctime.c in rts.src

Description The asctime function converts a broken-down time into a string with the following form:

```
Mon Jan 11 11:18:36 1988 \n\0
```

The function returns a pointer to the converted string.

For more information about the functions and types that the time.h header declares and defines, see section 7.3.13, *Time Functions (time.h)*, on page 7-21.

asin — *Arc Sine*

Syntax

#include <math.h>

double asin(double x**);**

Defined in

asin.c in rts.src

Description

The asin function returns the arc sine of a floating-point argument x, which must be in the range [–1, 1]. The return value is an angle in the range [–π/2, π/2] radians.

Example

```
double realval, radians;
realval = 1.0;
radians = asin(realval);  /* asin returns π/2 */
```

assert — *Insert Diagnostic Information Macro*

Syntax

#include <assert.h>

void assert(int expr**);**

Defined in

assert.h as macro

Description

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

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

The header file that defines the assert macro refers to another macro, NDEBUG. If you have defined NDEBUG as a macro name when the assert.h header is included in the source file, the assert macro is defined as:

```
#define assert(ignore)
```

Example

In this example, an integer i is divided by another integer j. Since dividing by 0 is an illegal operation, the example uses the assert macro to test j before the division. If j = = 0, assert issues a message and aborts the program.

```
int   i, j;
assert(j);
q = i/j;
```

atan — *Polar Arc Tangent*

Syntax

#include <math.h>

double atan(double x**);**

Defined in

atan.c in rts.src

Description

The atan function returns the arc tangent of a floating-point argument x. The return value is an angle in the range [–π/2, π/2] radians.

Example

```
double realval, radians;

realval = 0.0;
radians = atan(realval);      /* return value = 0 */
```

atan2 — *Cartesian Arc Tangent*

Syntax

#include <math.h>

double atan2(double y, double x**);**

Defined in

atan2.c in rts.src

Description

The atan2 function returns the inverse 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

```
double rvalu, rvalv;
double radians;

rvalu  = 0.0;
rvalv  = 1.0;
radians = atan2(rvalr, rvalu);   /* return value = 0 */
```

atexit — *Register Function Called by Exit ()*

Syntax

#include <stdlib.h>

int 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, the functions that were registered are called without arguments in reverse order of their registration.

atof/atoi/atol — *Convert String to Number*

Syntax

#include <stdlib.h>

double atof(const char *st**);**
int atoi(register const char *st**);**
long atol(register const char *st**);**

Defined in

atof.c, atoi.c, and atol.c in rts.src

Description

Three functions convert strings to numeric representations:

- The atof function converts a string into a floating-point value. Argument st points to the string; the string must have the following format:

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

- The atoi function converts a string into an integer. Argument st points to the string; the string must have the following format:

  [*space*] [*sign*] *digits*

- The atol function converts a string into a long integer. Argument st points to the string; the string must have the following format:

  [*space*] [*sign*] *digits*

The *space* is indicated by a space (character), a horizontal or vertical tab, a carriage return, a form feed, or a new line. Following the *space* is an optional *sign*, and the *digits* that represent the integer portion of the number. The fractional part of the number follows, then the exponent, including an optional *sign*.

The first character that cannot be part of the number terminates the string.

The functions do not handle any overflow resulting from the conversion.

bsearch — *Array Search*

Syntax

#include <stdlib.h>

void ***bsearch(**register const void *key, register 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 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

calloc — *Allocate and Clear Memory*

Syntax

#include <stdlib.h>

void ***calloc(**size_t num, size_t size**);**

Defined in

memory.c in rts.src

Description

The calloc function allocates size bytes (size is an unsigned integer or size_t) for each of num 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. The constant _ _SYSMEM_SIZE defines the size of the heap as 2K bytes. You can change this amount at link time by invoking the linker with the –heap option and specifying the desired size of the heap (in bytes) directly after the option. (See section 6.1.4, *Dynamic Memory Allocation*, on page 6-6.)

Example

This example uses the calloc routine to allocate and clear 20 bytes.

```
prt = calloc (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.

Example

```
extern double ceil();
double answer;
answer = ceil(3.1415);   /* answer = 4.0  */
answer = ceil(–3.5);     /* answer = –3.0 */
```

clearerr — *Clear EOF and Error Indicators*

Syntax

#include <stdio.h>

void clearerr(FILE *_fp**);**

Defined in

clearerr.c in rts.src

Description

The clearerr functions clears the EOF and error indicators for the stream that _fp points to.

clock — *Processor Time*

Syntax

#include <time.h>

clock_t clock(void**);**

Defined in

clock.c in rts.src

Description

The clock function determines the amount of processor time used. It returns an approximation of the processor time used by a program since the program began running. The time in seconds is the return value divided by the value of the macro CLOCKS_PER_SEC.

If the processor time is not available or cannot be represented, the clock function returns the value of [(clock_t) –1].

Note: Writing Your Own Clock Function

The clock function is host-system specific, so you must write your own clock function or use the clock function supplied with the simulator. You must also define the CLOCKS_PER_SEC macro according to the units of your clock so that the value returned by clock()—number of clock ticks—can be divided by CLOCKS_PER_SEC to produce a value in seconds.

For more information about the functions and types that the time.h header declares and defines, see section 7.3.13, *Time Functions (time.h)*, on page 7-21.

cos — *Cosine*

Syntax

#include <math.h>

double cos(double x**);**

Defined in

cos.c in rts.src

Description

The cos function returns the cosine of a floating-point number x. The angle x is expressed in radians. An argument with a large magnitude might produce a result with little or no significance.

Example

```
double radians, cval; /* cos returns cval    */
radians = 3.1415927;
cval = cos(radians);  /* return value = –1.0 */
```

cosh — *Hyperbolic Cosine*

Syntax

#include <math.h>

double cosh(double x**);**

Defined in

cosh.c in rts.src

Description

The cosh function returns the hyperbolic cosine of a floating-point number x. A range error occurs if the magnitude of the argument is too large.

Example

```
double x, y;

x = 0.0;
y = cosh(x); /* return value = 1.0 */
```

ctime — *Calendar Time*

Syntax

#include <time.h>

char *ctime(const time_t *timer**);**

Defined in

ctime.c in rts.src

Description

The ctime function converts a calendar time (pointed to by timer) to local time in the form of a string. This is equivalent to:

```
asctime(localtime(timer))
```

The function returns the pointer returned by the asctime function.

For more information about the functions and types that the time.h header declares and defines, see section 7.3.13, *Time Functions (time.h)*, on page 7-21.

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 expresses seconds.

For more information about the functions and types that the time.h header declares and defines, see section 7.3.13, *Time Functions (time.h)*, on page 7-21.

div/ldiv — *Division*

Syntax

#include <stdlib.h>

div_t div(register int numer, register int denom**);**
ldiv_t ldiv(register long numer, register long 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;
```

The sign of the quotient is negative if either but not both of the operands is negative. The sign of the remainder is the same as the sign of the dividend.

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. The exit function can accept EXIT_FAILURE as a value. (See the abort function on page 7-31).

You can modify the exit function to perform application-specific shut-down tasks. The unmodified function simply runs in an infinite loop until the system is reset.

The exit function cannot return to its caller.

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 = 7.38, which is e**2.0 */
```

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.

Example

```
double x, y;

x = –57.5;
y = fabs(x);       /* return value = +57.5 */
```

fclose — *Close File*

Syntax

#include <stdio.h>

int fclose(FILE *_fp**);**

Defined in

fclose.c in rts.src

Description

The fclose function flushes the stream that _fp points to and closes the file associated with that stream.

feof — *Test EOF Indicator*

Syntax

#include <stdio.h>

int feof(FILE *_fp**);**

Defined in

feof.c in rts.src

Description

The feof function tests the EOF indicator for the stream pointed to by _fp.

ferror — *Test Error Indicator*

Syntax

#include <stdio.h>

int ferror(FILE *_fp**);**

Defined in

ferror.c in rts.src

Description

The ferror function tests the error indicator for the stream pointed to by _fp.

fflush — *Flush I/O Buffer*

Syntax

#include <stdio.h>

int fflush(register FILE *_fp**);**

Defined in

fflush.c in rts.src

Description

The fflush function flushes the I/O buffer for the stream pointed to by _fp.

fgetc — *Read Next Character*

Syntax

#include <stdio.h>

int fgetc(register FILE *_fp**);**

Defined in

fgetc.c in rts.src

Description

The fgetc function reads the next character in the stream pointed to by _fp.

fgetpos — *Store Object*

Syntax

#include <stdio.h>

int fgetpos(FILE *_fp**,** fpos_t *pos**);**

Defined in

fgetpos.c in rts.src

Description

The fgetpos function stores the object pointed to by pos to the current value of the file position indicator for the stream pointed to by _fp.

fgets — *Read Next Characters*

Syntax

#include <stdio.h>

char *fgets(char *_ptr, register int _size, register FILE *_fp**);**

Defined in

fgets.c in rts.src

Description

The fgets function reads the specified number of characters from the stream pointed to by _fp. The characters are placed in the array named by _ptr. The number of characters read is _size –1.

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.

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 floating-point remainder of x divided by y. If y == 0, the function returns 0.

Example

```
double x, y, r;

x = 11.0;
y = 5.0;
r = fmod(x, y);        /* fmod returns 1.0 */
```

fopen — *Open File*

Syntax

#include <stdio.h>

FILE ***fopen(**const char *_fname, const char *_mode**);**

Defined in

fopen.c in rts.src

Description

The fopen function opens the file that _fname points to. The string pointed to by _mode describes how to open the file.

fprintf — *Write Stream*

Syntax

#include <stdio.h>

int fprintf(FILE *_fp, const char *_format, ...**);**

Defined in

fprint.c in rts.src

Description

The fprintf function writes to the stream pointed to by _fp. The string pointed to by _format describes how to write the stream.

fputc — *Write Character*

Syntax

#include <stdio.h>

int fputc(int _c, register FILE *_fp**);**

Defined in

fputc.c in rts.src

Description

The fputc function writes a character to the stream pointed to by _fp.

fputs — *Write String*

Syntax

#include <stdio.h>

int fputs(const char *_ptr, register FILE *_fp**);**

Defined in

fputs.c in rts.src

Description

The fputs function writes the string pointed to by _ptr to the stream pointed to by _fp.

fread — *Read Stream*

Syntax

#include <stdio.h>

size_t fread(void *_ptr, size_t _size, size_t _count, FILE *_fp**);**

Defined in

fread.c in rts.src

Description

The fread function reads from the stream pointed to by _fp. The input is stored in the array pointed to by _ptr. The number of objects read is _count. The size of the objects is _size.

free — *Deallocate Memory*

Syntax

#include <stdlib.h>

void free(void *packet**);**

Defined in

memory.c in rts.src

Description

The free function deallocates memory space (pointed to by packet) that was previously allocated by a malloc, calloc, or realloc call. This makes the memory space available again. If you attempt to free unallocated space, the function takes no action and returns. For more information, see section 6.1.4, *Dynamic Memory Allocation*, on page 6-6.

Example

This example allocates ten bytes and frees them.

```
char *x;
x = malloc(10);       /*  allocate 10 bytes   */
free(x);              /*  free 10 bytes       */
```

freopen — *Open File*

Syntax

#include <stdio.h>

FILE ***freopen(**const char *_fname, const char *_mode, register FILE *_fp**);**

Defined in

fopen.c in rts.src

Description

The freopen function opens the file pointed to by _fname, and associates with it the stream pointed to by _fp. The string pointed to by _mode describes how to open the file.

frexp — *Fraction and Exponent*

Syntax

#include <math.h>

double frexp(double value**,** int *exp**)**;

Defined in

frexp.c in rts.src

Description

The frexp function breaks a floating-point number into a normalized fraction and the integer power of 2. The function returns a value 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 */
```

fscanf — *Read Stream*

Syntax

#include <stdio.h>

int fscanf(FILE *_fp, const char *_fmt, ...**);**

Defined in

fscanf.c in rts.src

Description

The fscanf function reads from the stream pointed to by _fp. The string pointed to by _fmt describes how to read the stream.

fseek — *Set File Position Indicator*

Syntax

#include <stdio.h>

int fseek(register FILE *_fp, long _offset, int _ptrname**);**

Defined in

fseek.c in rts.src

Description

The fseek function sets the file position indicator for the stream pointed to by _fp. The position is specified by _ptrname. For a binary file, use _offset to position the indicator from _ptrname. For a text file, offset must be 0.

fsetpos — *Set File Position Indicator*

Syntax

```
#include   <stdio.h>

int fseek(FILE *_fp, const fpos_t *_pos);
```

Defined in fsetpos.c in rts.src

Description The fsetpos function sets the file position indicator for the stream pointed to by _fp to _pos. The pointer _pos must be a value from fgetpos() on the same stream.

ftell — *Get Current File Position Indicator*

Syntax

```
#include   <stdio.h>

long ftell(FILE *_fp);
```

Defined in ftell.c in rts.src

Description The ftell function gets the current value of the file position indicator for the stream pointed to by _fp.

fwrite — *Write Block of Data*

Syntax

```
#include   <stdio.h>

size_t fwrite(const void *_ptr, size_t _size, size_t _count, register FILE *_fp);
```

Defined in fwrite.c in rtd.src

Description The fwrite function writes a block of data from the memory pointed to by _ptr to the stream that _fp points to.

getc — *Read Next Character*

Syntax

```
#include   <stdio.h>

int getc(FILE *_fp);
```

Defined in fgetc.c in rts.src

Description The getc function reads the next character in the file pointed to by _fp.

getchar — *Read Next Character From Standard Input*

Syntax

#include <stdio.h>

int getchar(void**);**

Defined in

fgetc.c in rts.src

Description

The getchar function reads the next character from the standard input device.

getenv — *Get Environment Information*

Syntax

#include <stdlib.h>

char *getenv(const char *_string**);**

Defined in

trgdrv.c in rts.src

Description

The getenv function returns the environment information for the variable associated with _string.

gets — *Read Next From Standard Input*

Syntax

#include <stdio.h>

char *gets(char *_ptr**);**

Defined in

fgets.c in rts.src

Description

The gets function reads an input line from the standard input device. The characters are placed in the array named by _ptr.

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 a broken-down time, which is expressed as Greenwich Mean Time.

For more information about the functions and types that the time.h header declares and defines, see section 7.3.13, *Time Functions (time.h)*, on page 7-21.

isxxx *Character Typing*

Syntax

#include <ctype.h>

int isalnum(int c**);**
int isalpha(int c**);**
int isascii(int c**);**
int iscntrl(int c**);**
int isdigit(int c**);**
int isgraph(int c**);**
int islower(int c**);**
int isprint(int c**);**
int ispunct(int c**);**
int isspace(int c**);**
int isupper(int c**);**
int isxdigit(int c**);**

Defined in

is*xxx*.c and ctype.c in rts.src
Also defined in ctype.h as macros

Description

These functions test a single argument, c, to see if it is a particular type of character — alphabetic, alphanumeric, numeric, ASCII, etc. If the test is true, the function returns a nonzero value; if the test is false, the function returns 0. 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 (any character from 0–127)
iscntrl	Identifies control characters (ASCII characters 0–31 and 127)
isdigit	Identifies numeric characters between 0 and 9 (inclusive)
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 tab (horizontal or vertical), space bar, carriage return, form feed, and new line 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.

Chapter 8

Library-Build Utility

When using the C compiler, you can compile your code under a number of different configurations and options that are not necessarily compatible with one another. Since it would be cumbersome to include all possible combinations in individual run-time-support libraries, this package includes the source archive, rts.src, which contains all run-time-support functions.

You can build your own run-time-support libraries by using the mk27 utility described in this chapter and the archiver described in the *TMS320C27xx Assembly Language Tools User's Guide*.

Topic **Page**

8.1 Invoking the Library-Build Utility

The general syntax for invoking the library utility is:

mk27 [*options*] *src_arch1* [**–l***obj.lib1*] [*src_arch2* [**–l***obj.lib2*]] ...

mk27 is the command that invokes the utility.

options can appear anywhere on the command line or in a linker command file. (Options are discussed below and in section 8.2, *Options Summary*.)

src_arch is the name of a source archive file. For each source archive named, mk27 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, mk27 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.

The mk27 utility runs the shell program on each source file in the archive to compile and/or assemble it. Then, the utility collects all the object files into the object library. All the tools must be in your PATH environment variable. The utility ignores the environment variables TMP, C_OPTION, and C_DIR.

8.1.1 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, and linker. 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 mk27 to use header files contained in the source archive and leave them in the current directory after the utility has completed execution. You can use this option to install the run-time-support header files from the rts.src archive that is shipped with the tools.

––k Instructs mk27 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 mk27 to suppress header information (quiet).

––u Instructs mk27 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 run-time-support functions to suit your application.

––v Prints progress information to the screen during execution of the utility. Normally, the utility operates silently (no screen messages).

8.2 Options Summary

The other options that can be used with the library-build utility correspond directly to the options that the compiler uses. These options are described in detail in section 2.2, *Changing Compiler Behavior With Compiler Options*, on page 2-6. The following table provides a summary of the options that you can use with the utility.

Table 8–1. Options Summary

(a) Options that control the compiler/shell

General Options	Effect
–g	Enable symbolic debugging

(b) Options that control the parser

Parser Options	Effect
–pg	Enable trigraph expansion
–pk	Make code K&R compatible
–pw0	Disables all warning messages
–pw1	Enables serious warning messages (default)
–pw2	Enables all warning messages

(c) Options that control the optimization level

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 mk27 automatically sets –oI0 and –op0.
–ox (equivalent to –x2)	Define _INLINE + above + invoke optimizer (at –o2 if not specified differently)

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

Inlining Options	Effect
–x1	Enable intrinsic function inlining
–x2 (or –x)	Define _INLINE + above + invoke optimizer (at –o2 if not specified differently)

Table 8–1. Options Summary (Continued)

(e) Options that are machine specific

Runtime Model Options	Effect
–ma	Assume aliased variables
–mn	Enable optimization disabled by –g
–mf	Optimize for speed instead of for space

(f) Options that overlook type checking

Type Checking Options	Effect
–tf	Relax prototype checking
–tp	Relax pointer combination checking

(g) Option that controls the assembler

Assembler Options	Effect
–as	Keep labels as symbols

(h) Options that change the default file extensions

Default File Extension Options	Effect
–ea<*.ext*>	Extension for assembly files (default is .asm)
–eo<*.ext*>	Extension for object files (default is .obj)

Appendix A

Invoking the Compiler Tools Individually

The TMS320C27xx C compiler offers you the versatility of invoking all of the tools at once, using the shell, or invoking each tool individually. To satisfy a variety of applications, you can invoke the compiler (parser and code generator), the assembler, and the linker as individual programs. This section also describes how to invoke the interlist utility outside the shell.

Topic **Page**

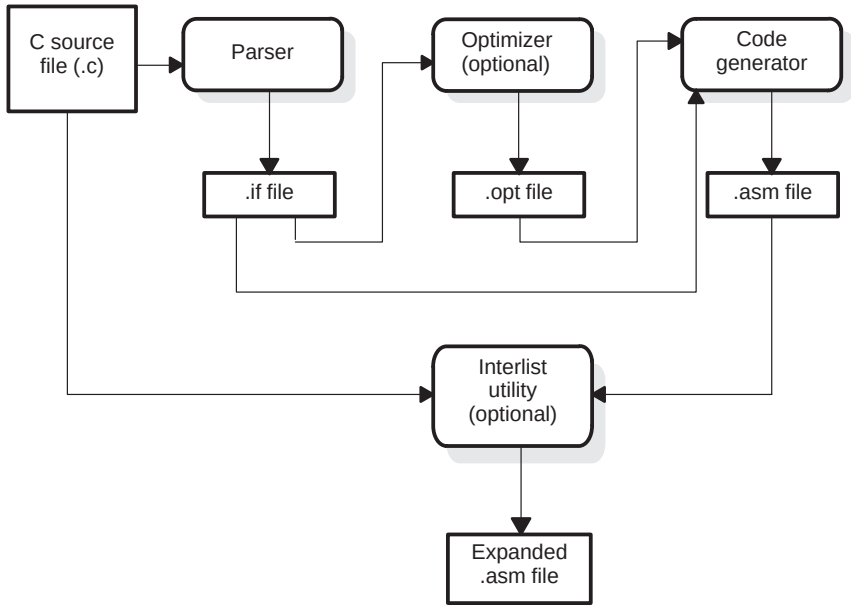
A.1 Which Tools Can be Invoked Individually

To satisfy a variety of applications, you can invoke the compiler, the assembler, and the linker as individual programs.

A.1.1 About the Compiler

The compiler is made up of three distinct programs: the parser, the optimizer, and the code generator. The compiler can also invoke the interlist utility.

Figure A–1. Compiler Overview



A.1.1.1 About the Parser

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. Section A.2, *Invoking the Parser individually*, on page A-4 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.

A.1.1.2 About the Optimizer

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 3.1, *Using the C Compiler Optimizer*, on page 3-2 describes the optimizer.

A.1.1.3 About the Code Generator

The input for the code generator is the intermediate file produced by the parser (.if) or the optimizer (.opt). The code generator produces an assembly language source file. Section A.5, *Invoking the Code Generator*, on page A-9 describes how to run the code generator.

A.1.1.4 About the Interlist Utility

The inputs for the interlist utility 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, *Using the Interlist Utility*, on page 2-28 and section A.6, *Invoking the Interlist Utility*, on page A-11 describe the use of the interlist utility.

A.1.2 About the Assembler and Linker

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 *TMS320C27xx Assembly Language Tools User's Guide.*

The input for the linker is the COFF object file produced by the assembler. The linker produces an executable object file. Chapter 4, *Linking C Code,* describes how to run the linker. The linker is described fully in the *TMS320C27xx Assembly Language Tool User's Guide.*

A.2 Invoking the Parser Individually

The first step in compiling a 'C27xx 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 or the optimizer. To invoke the parser, enter the following:

ac27 *input file* [*output file*] [*options*]

ac27 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 do not supply a filename for the output file, the parser uses the input filename with an extension of *.if*.

options affect parser operation. Each option available for the standalone parser has a corresponding shell option that performs the same function. Table A–1 shows the parser options, the shell options, and the corresponding functions.

Note: Using Wildcards

Using wildcards will only take one file, not multiple files.

Table A–1. Parser Options

Parser Option	Function	See Shell Option	See Page
–d*name* [=def]	Predefines macro *name*	–d*name* [=def]	2-11
–e	Treats code-E errors as warnings	–pe	2-30
–i*dir*	Defines #include search path	–i*dir*	2-12, 2-20
–k	Allows K&R compatibility	–pk	5-16
–l (lowercase L)	Generates .pp file†	–pl	2-21
–n	Suppresses #line directives	–pn	2-21
–o	Preprocesses only	–po	2-21
–q	Suppresses progress messages (quiet)	–q	2-13
–r	Generates error listing	–pr	2-30
–tf	Relaxes prototype checking	–tf	2-17
–tp	Relaxes pointer combination	–tp	2-17
–u*name*	Undefines macro *name*	–u*name*	2-14
–w	Suppresses warning messages	–pw	2-30
–x	Enables inlining of user functions (implies –o2)	–x2	2-24
–x0	Disables function inlining	–x0	2-24
–?	Enables trigraph expansion	–pg	2-22

† When running ac27 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.

A.3 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 such 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.

   ```
   cl27 –po file.c
   ```

   If you want to use the –d, –u, or –i options, use them on this first pass. This pass produces a preprocessed output file called file.pp. For more information about the preprocessor, see section 2.3, *Controlling the Preprocessor*, on page 2-19.

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

   ```
   cl27 file.pp
   ```

   You can use any other options on this final pass.

A.4 Invoking the Optimizer Individually

The second step in compiling a 'C27xx 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 optimized 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:

opt27 [*input file* [*output file*]] [*options*]

opt27 is the command that invokes the optimizer.

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

output file names the intermediate file that the optimizer creates. If you do not supply a filename for the output file, the optimizer uses the input filename with an extension of *.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 the shell. Table A–2 on page A-8 shows the optimizer options, the shell options, and the corresponding functions.

Table A–2. Optimizer Options and Shell Options

		See	
Optimizer Option	**Function**	**Shell Option**	**Page**
–a	Assumes variables are aliased	–ma	2-8
–h*n*	Controls assumptions about library function calls	–ol*n*	3-4
–i*nn*	Sets automatic inlining size threshold (–o3 only)	–oi*size*	3-12
–k	Allows K&R compatibility	–pk	5-16
–n*n*	Generates optimization information file (–o3 only)	–on*n*	3-5
–o0	Optimizes at level 0 (register optimization)	–o0	3-2
–o1	Optimizes at level 1 (level 0 plus local optimization)	–o1	3-3
–o2	Optimizes at level 2 (level 1 plus global optimization)	–o2	3-3
–o3	Optimizes at level 3 (level 2 plus file optimization)	–o3	3-3
–q	Suppresses progress messages (quiet)	–q	2-13
–s	Interlists C source	–s	2-13, 2-28

† The –g option tells the optimizer that the register named is reserved for global use.

A.5 Invoking the Code Generator Individually

The third step in compiling a 'C27xx C program is to invoke the code generator. The code generator converts the intermediate file produced by the parser or the optimizer into an assembly language source file. You can modify this output file or use it as input for the assembler. The code generator produces reentrant relocatable code, which, after assembling and linking, can be stored in ROM.

To invoke the code generator as a standalone program, enter:

cg27 [*input file* [*output file* [*tempfile*]]] [*options*]

cg27 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 source 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 of *.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 shell option that performs the same function. The following table shows the code generator options, the shell options, and the corresponding functions.

Table A–3. Code Generator Options and Shell Options

Code Generator Options	Function	See Shell Options	See Page
–a	Assumes variables are aliased	–ma	2-8
–n	Reenables optimizations disabled by symbolic debugging	–mn	2-8
–o	Enables C source level debugging	–g†	2-11
–q	Suppresses progress messages (quiet)	–q	2-13
–s	Optimizes for space instead of for speed	–mf	2-8
–z‡	Retains the input file	—	—

† The –g option tells the code generator that the register named is reserved for global use
‡ The –z option tells the code generator to retain the input file (the intermediate file created by the parser or the optimizer). If you do not specify the –z option, the intermediate file is deleted.

A.6 Invoking the Interlist Utility Individually

Note: Interlisting With the Shell Program and the Optimizer

You can create an interlisted file by invoking the shell program with the –s option. Anytime that you request interlisting on optimized code, the optimizer, not the interlist utility, performs the interlist function.

The fourth step in compiling a 'C27xx 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 names the assembly language file produced by 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 compiler 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:

```
cl27 –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.

Appendix B

Glossary

A

ANSI (American National Standards Institute): A board that approves American National Standards.

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

aliasing: The ability for a single object to be accessed in more than one way, such as when two pointers point to a single object. It can disrupt optimization because any indirect reference could potentially refer to any other object.

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.

argument block: The part of the local frame used to pass arguments to other functions.

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

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: An extra entry that a symbol may have in the symbol table. Th entry contains additional information about the symbol (whether the symbol is a filename, a section name, a function name, and so on).

B

banner: Information in a compiler listing that denotes one pass through the compiler. The information identifies the compiler.

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

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

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

byte: Traditionally, a sequence of eight adjacent bits operated upon as a unit. However, the 'C27xx byte is 16 bits.

Note: 'C27xx 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 'C27xx 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). 'C27xx 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 optimizer and produces an assembly language source file.

command file: A file that contains options, filenames, directives, or comments for the compiler or 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.

data memory: A section of memory that contains external variables, static variables, and the system stack.

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 run time. This is accomplished by declaring a large memory pool, or heap, and then using the functions to allocate memory from the heap.

E

emulator: A hardware development system that accepts the same inputs and produces the same outputs as the 'C27xx device.

entry point: The location in target memory where the program begins execution.

executable module: An object file that has been linked and can be run 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 ’C27xx, 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.

function inlining: Code for a function is inserted at the point of the call. Function inlining saves the overhead (the code necessary to enter and exit the function) of a function call. Function inlining also allows the optimizer to make the function code as efficient as possible in the context of the surrounding code.

G

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

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.

hole: An area between the input sections that compose an output section that contains no 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 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: A preprocessor that is combined with the parser, allowing for faster compilation.

interlist utility: A utility that includes (as comments) your original C source statements with the assembly language output from the assembler.

K

K&R: Kernighan and Ritchie C, the de facto standard as defined in the second edition of *The C Programming Language*. 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 assembler source statement and corresponds to the address of that statement. This is the only assembler statement that can begin in column 1.

line number entry: An item 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 section program counter (SPC).

loader: A device that places an executable module into system memory.

M

macro: A user-defined routine that is 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.

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 the symbols were defined.

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

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.

object module: A group of object files that have been linked together to create an executable program.

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

options: Command-line 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 by rewriting pieces of code to take advantage of the 'C27xx architecture.

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. 'C27xx 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 preprocessor functions, checks the syntax, and produces an intermediate file that can be used as input for the optimizer or code generator.

partial linking: The linking of a file that will be linked again.

pragma directive: A pragma directive tells the preprocessor how to treat functions.

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

program memory: A section of memory that contains executable code.

R

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

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

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

runtime environment: The conditions within which the system operates. These include memory and register conventions, stack organization, function call conventions, and system initialization.

run-time-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.

run-time-support library: A library file, rts.src, that contains the source for the run-time-support functions as well as for other functions and routines.

S

section: A relocatable block of code or data that will 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: A process that fills 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 memory location within a section; each section has its own SPC.

static variable: 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; their previous value is resumed when the function or program is reentered.

storage class: An entry in the symbol table that indicates how a symbol must be accessed.

string table: A matrix 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.

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

T

target memory: The physical memory in a 'C27xx-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 variable: A variable that may hold (at different times) objects of different types and sizes.

unsigned value: 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 sequence of bits or characters that is stored, addressed, transmitted, and operated on as a unit. A 16-bit addressable location in target memory.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z