TMS320C28x
Optimizing C/C++ Compiler
User's Guide

Preliminary

Literature Number: SPRU514
August 2001

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Read This First

About This Manual

The *TMS320C28x Optimizing C/C++ Compiler User's Guide* explains how to use these compiler tools:

❑ Compiler
❑ Post-link optimizer
❑ Library-build utility
❑ Interlist utility
❑ C++ name demangling utility

The TMS320C28x C/C++ compiler accepts American National Standards Institute (ANSI) standard C and C++ source code and produces assembly language source code for the TMS320C28x devices.

This user's guide discusses the characteristics of the C/C++ compiler. It assumes that you already know how to write C/C++ programs. *The C Programming Language* (second edition), by Brian W. Kernighan and Dennis M. Ritchie, describes C based on the ANSI C standard. You can use the Kernighan and Ritchie (hereafter referred to as K&R) book as a supplement to this manual. References to K&R C (as opposed to ANSI C) in this manual refer to the C language as defined in first edition of Kernighan and Ritchie's *The C Programming Language*.

The C++ Programming Language (third edition), by Bjarne Stroustrup describes C++ based on the ANSI standard. For more information about supported and unsupported C++ language features, see section 6.2, Characteristics of TMS320C28x C++, on page 6-5.

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:

  cl2000 –v28 [*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:

  ac2000*inputfile* [*outputfile*] [*options]*

  The ac2000 command has three parameters. The first parameter, *input-file*, 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:

  ```
  lnk2000 {–c | –cr} filenames [–o name.out] –l libraryname
  ```

- The TMS320C2800 core is referred to as TMS320C28x and C28x.

Related Documentation From Texas Instruments

The following books describe the TMS320C28x and related support tools. To obtain any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, identify the book by its title and literature number (located on the title page):

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

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

TMS320C28x Assembly Language Tools User’s Guide (literature number SPRU513) 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 TMS320C28x™ device.

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

Related Documentation

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

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

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

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

Programming Language C++, ISO/IEC 14882–1998, American National Standards Institute (ANSI standard for C++)

The Annotated C++ Reference Manual, Margaret A. Ellis and Bjarne Stroustrup, published by Addison-Wesley Publishing Company, Reading, Massachusetts, 1990

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

The C++ Programming Language (third edition), Bjarne Stroustrup, published by Addison-Wesley Publishing Company, Reading, Massachusetts, 1997

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

Trademarks

Contents

Figures

Tables

Examples

Notes

Chapter 1

Introduction

The TMS320C28x™ is fully supported by a complete set of code generation tools, including an optimizing C/C++ compiler, assembler, linker, and assorted utilities.

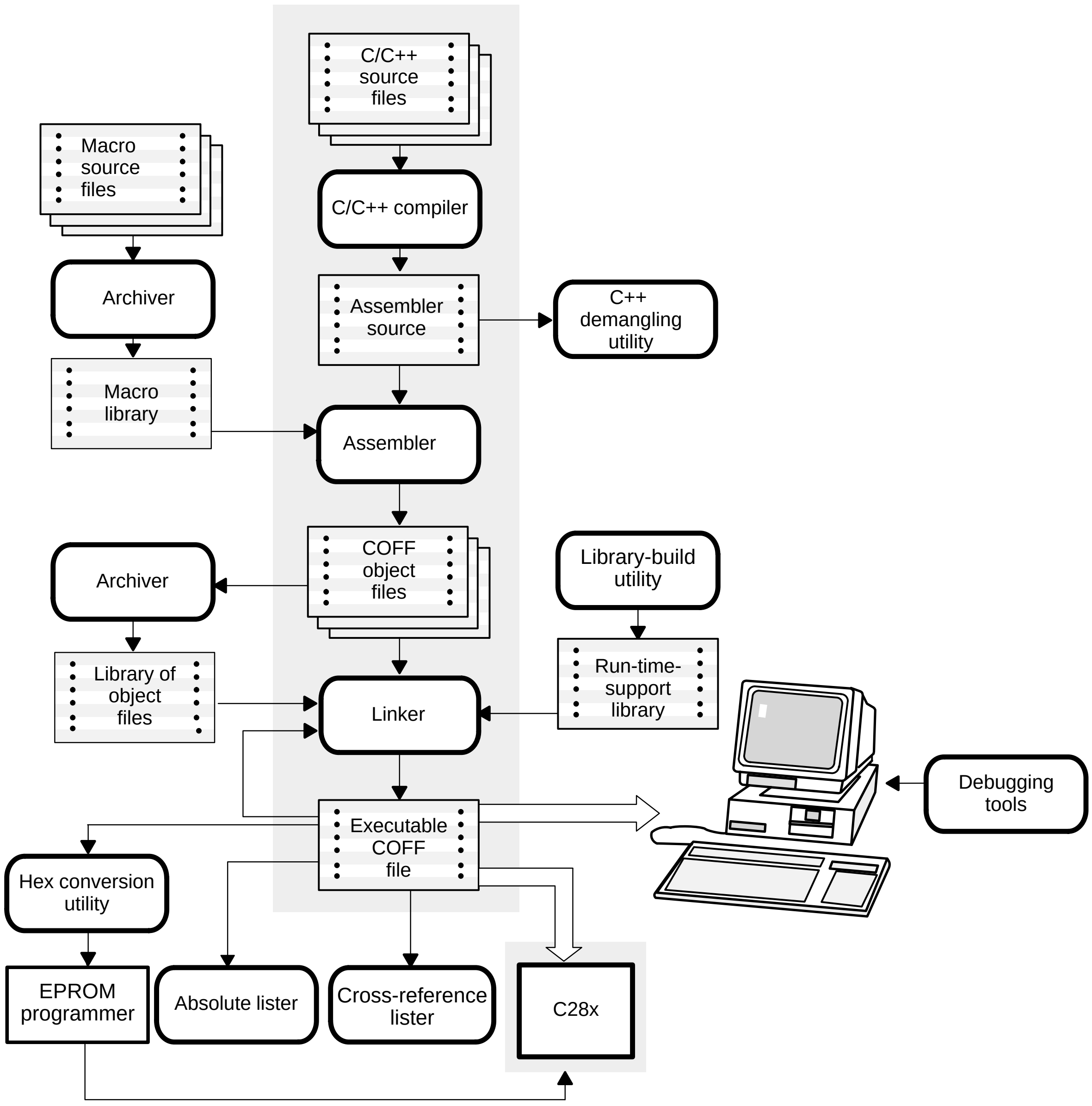
This chapter provides an overview of these tools and introduces the features of the optimizing C/C++ compiler. The assembler and linker are described in detail in the *TMS320C28x Assembly Language Tools User’s Guide*.

Topic **Page**

1.1 Software Development Tools Overview

Figure 1–1 illustrates the TMS320C28x software development flow. The shaded portion of the figure highlights the most common path of software development for C/C++ language programs. The other portions are peripheral functions that enhance the development process.

Figure 1–1. TMS320C28x Software Development Flow



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

- The **C/C++ compiler** accepts C/C++ source code and produces C28x 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/C++ programs.
  - The interlist utility incorporates C/C++ source statements with assembly language output.

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

- The **C++ name demangling utility** (dem2000)is a debugging aid that converts names mangled by the compiler back to their original names as declared in the C++ source. As shown in Figure 1–1, you can use the C++ name demangling utility on the assembly file that is output by the compiler; you can also use this utility on the assembler listing file and the linker map file. See Chapter 10, *C++ Name Demangler*, for more information.

- The **assembler** (asm2000 –v28) translates assembly language source files into machine language object files. The machine language is based on common object file format (COFF). The *TMS320C28x Assembly Language Tools User's Guide* explains how to use the assembler.

- The **linker** (lnk2000) 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 *TMS320C28x Assembly Language Tools User's Guide* explains the linker in detail.

- The **archiver** (ar2000) 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/C++ compiler. The *TMS320C28x Assembly Language Tools Guide* explains how to use the archiver.

- ❑ You can use the **library-build utility** (mk2000 –v28) to build 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 for ANSI C/C++ library source. ANSI C/C++ library object code for the run-time-support functions is collected in rts.lib.

  The **run-time-support libraries** contain the ANSI C standard run-time-support functions, C++ library, compiler-utility functions, and floating-point arithmetic functions that are supported by the C28xcompiler. See Chapter 8, *Run-Time-Support Functions*, for more information.

- ❑ Code Composer Studio accepts executable COFF files as input, but most EPROM programmers do not. The **hex conversion utility** (hex2000) 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 *TMS320C28x Assembly Language Tools User's Guide* explains how to use the hex conversion utility.

- ❑ The **absolute lister** (abs2000) 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 *TMS320C28x Assembly Language Tools User's Guide*.

- ❑ The **cross-reference lister** (xref2000) uses object files to produce a cross-reference listing showing symbols, their definitions, and their references across all linked source files. The *TMS320C28x Assembly Language Tools User's Guide* explains how to use the cross-reference utility.

- ❑ The **post-link optimizer** (plink2000) removes or modifies assembly language instructions to generate better code. The post-link optimizer must be run with the shell option –plink. See Chapter 5, *Post-Link Optimizer*, for more information.

The purpose of this development process is to produce a module that can be executed in a C28x target system. You can use a simulator or emulator to refine and correct your code.

1.2 C/C++ Compiler Overview

The TMS320C28x C/C++ compiler is a full-featured optimizing compiler that translates standard ANSI C and C++ programs into C28x assembly language source. The following sections describe key characteristics of the compiler.

1.2.1 C/C++ Language Features

❑ **ANSI standard C**

The TMS320C28x compiler 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 C++**

The TMS320C28x compiler supports C++ as defined by the ISO/IEC 14882–1998 standard with certain exceptions. For more information, see Chapter 6, *TMS320C28xC/C++ Language Implementation*.

❑ **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. The library is also extended to include versions of the rts functions that support accessing far memory. The C++ library includes the ANSI C subset as well as those components necessary for language support. For more information see Chapter 8, *Run-Time-Support Functions*.

1.2.2 Output Files

The following features pertain to output files created by the compiler:

- **Assembly source output**

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

1.2.3 Compiler Interface

The following features pertain to interfacing with the compiler:

- **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/C++ functions that call each other. For more information, see Chapter 7, *Runtime Environment*.

1.2.4 Compiler Operation

The following features pertain to the operation of the compiler:

- **Integrated preprocessor**

  The C/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 2.5, *Controlling the Preprocessor*, on page 2-23.

- **Optimization**

  The compiler uses a sophisticated optimization pass that employs several advanced techniques for generating efficient, compact code from C/C++ source. General optimizations can be applied to any C/C++ code. C28x-specific optimizations take advantage of the features unique to the C28x architecture. For more information about the C/C++ compiler's optimization techniques, see Chapter 3, *Optimizing Your Code*.

1.2.5 Utilities

The following features pertain to the compiler utilities:

- **Source interlist utility**

  The compiler package includes a utility that interlists your original C/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/C++ statement. For more information, see section 2.10, *Using the Interlist Utility*, on page 2-39.

- **Library-build utility**

  The compiler package has a utility (mk2000 –v28) which can create object libraries from archived source libraries in one step. This is useful for re-compiling the RTS library using compiler options that fit your needs. The complete RTS library source is included with the compiler product.

- **C++ name demangler utility**

  The C++ name demangler (dem2000) is a debugging aid that converts mangled names found in the output of the compiler tools (such as assembly files and linker error messages) back to their original name as declared in C++ source. For more information, see Chapter 10, *C++ Name Demangler Utility.*

Chapter 2

Using the C/C++ Compiler

Translating your source program into code that the C28x can execute is a multistep process. You must compile, assemble, and link your source files to create an executable object file. The C28x 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.

Topic **Page**

2.1 About the Shell Program

The compiler shell program (cl2000 –v28) lets you compile, assemble, and optionally link in one step. The shell runs one or more source modules through the following tools:

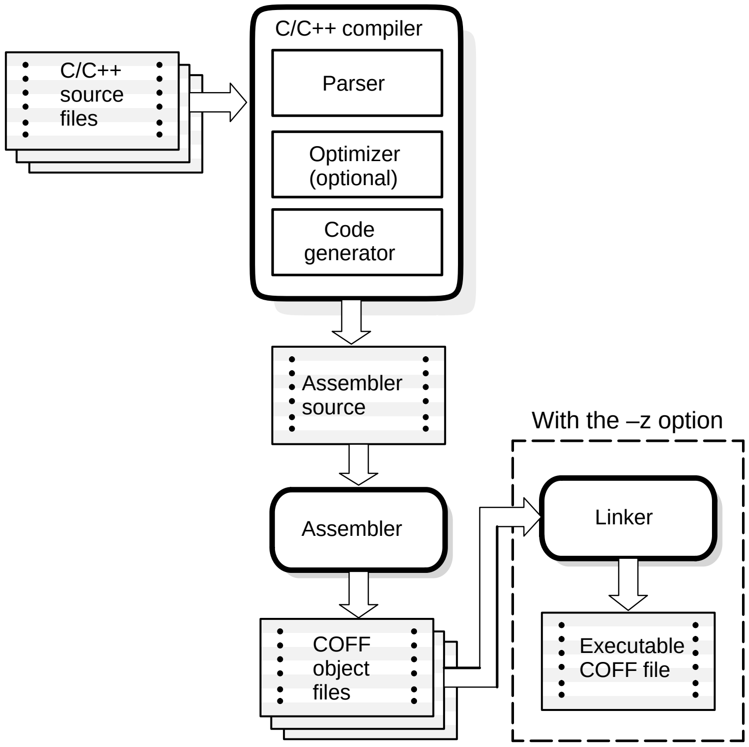
1) The **compiler**, which includes the parser, optimizer, and code generator, accepts C/C++ source code and produces C28x assembly language source code.

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/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 *TMS320C28x Assembly Language Tools User’s Guide*.

2.2 Invoking the C/C++ Compiler Shell

To invoke the compiler shell, enter:

cl2000 –v28 [*–options*] *filenames* [*object files*] [**–z** [*link_options*]]

cl2000 –v28	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 2-6.)
filenames	One or more C/C++ source files or assembly language source files
object files	Name of the additional object files for the linking process
–z	Option that invokes the linker. See Chapter 4, *Linking C/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:

```
cl2000 –v28 –q symtab file seek.asm
```

As cl2000 –v28 encounters each source file, it prints the C/C++ filenames in square brackets ([ ]), assembly language filenames in angle brackets (< >). This example uses the –q option to suppress the additional progress information that cl2000 –v28 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:

```
%cl2000 –v28 symtab
[symtab.c]
TMS320C2000 ANSI C/C++ Compiler Version XX
Copyright (c) 1996–2001 Texas Instruments Incorporated

   “symtab.c”   ==> symtab
TMS320C2000 ANSI C Codegen      Version XX
Copyright (c) 1996–2001 Texas Instruments Incorporated

   “symtab.c”   ==> symtab
TMS320C2000 Assembler           Version XX
Copyright (c) 1996–2001 Texas Instruments Incorporated

PASS1
PASS2
```

2.3 Changing the Compiler's Behavior With Options

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

The following apply to the compiler options:

- ❑ Options are either single letters or 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, –al and –as can be combined as –als.
- ❑ 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. For more information, see section 2.4, Changing the Compiler's Behavior With Environment Variables, on page 2-20.

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 **cl2000 –v28** 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	2-12
–abs	Generate absolute listing, must follow –z	2-12
–b	Generates a user information file	2-12
–c	Disables linking (negates –z)	2-12, 4-5
–d*name* [=*def*]	Predefines *name*	2-13
–g	Enables symbolic debugging	2-13
–i*directory*	Defines #include search path	2-13, 2-24
–k	Keeps the assembly language (.asm) file	2-13
–n	Compiles or assembly optimizes only	2-13
–plink	Performs post-link optimization, must follow the –z option	5-1
–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/C++ and assembly source statements	2-13
–ss	Interlists C/C++ source and assembly statements	2-14, 3-13
–u*name*	Undefines *name*	2-14
–z	Enables linking	2-14

Table 2–1. Shell Options Summary (Continued)

(b) Options that change the default file extensions

Option	Effect	Page
–ea[.]*extension*	Sets default extension for assembly source files	2-17
–eo[.]*extension*	Sets default extension for object files	2-17

(c) Options that specify files

Option	Effect	Page
–fa*filename*	Identifies *filename* as an assembly file, regardless of its extension. By default, the compiler treats .asm or .s* (extension begins with s) as an assembly file.	2-17
–fc*filename*	Identifies *filename* as a C file, regardless of its extension. By default, the compiler treats .c or no extension as a C file.	2-17
–fg	Compiles files with C extensions as C++ files	2-17
–fo*filename*	Identifies *filename* as an object file, regardless of its extension. By default, the compiler treats .o* as an object file.	2-17
–fp*filename*	Identifies *filename* as a C++ file, regardless of its extension. By default, the compiler treats .C, .cpp, .cc, or .cxx as a C++ file.†	2-17

† Case sensitivity in filename extensions is determined by your operating system. If your operating system is not case sensitive, .C is interpreted as a C file.

(d) Options that specify directories

Option	Effect	Page
–fb*directory*	Specifies an absolute listing file directory	2-18
–ff*directory*	Specifies an assembly listing file	2-18
–fr*directory*	Specifies object file directory	2-18
–fs*directory*	Specifies assembly file directory	2-18
–ft*directory*	Specifies temporary file directory	2-18

Table 2–1. Shell Options Summary (Continued)

(e) Options that are machine-specific

Option	Effect	Page
–ma	Assumes aliased variables	2-14, 3-11
–md	Disables DP load optimizations	2-14
–me	Disables generation of fast branch instructions	2-14
–mf	Optimizes for code speed over size	2-14
–mi	Disables generation of RPT instructions	2-14
–ml	Generates large memory model code	2-15, 6-18
–mn	Enables optimizer options disabled by –g	2-15
–ms	Optimizes for size over speed	2-15, 3-17
–mt	Generates code to access switch tables from data space and enables unified memory instructions	2-15
–mu	Encodes C2xlp OUT instructions as C28x UOUT instructions.	2-15
–mx	Expands assembly macros	2-15
–v28	Specifies TMS320C28x architecture	2-15

(f) Options that control parsing

Option	**Effect**	**Page**
–pe	Enables embedded C++	6-6
–pi	Disables definition-controlled inlining (but –o3 optimizations still perform automatic inlining)	2-37
–pk	Allows K&R compatibility	6-32
–pl *file*	Outputs raw listing information to *file*	2-33
–pm	Program mode compilation	3-6
–pn	Disables intrinsic functions	
–pr	Enables relaxed mode; ignores strict ANSI violations	6-35
–ps	Enables strict ANSI mode (for C/C++, not K&R C)	6-35
–px *file*	Outputs cross-reference information to *file*	2-32
–rtti	Enables run time type information (RTTI)	6-5

Table 2–1. Shell Options Summary (Continued)

(g) Options that control preprocessing

Option	Effect	Page
–ppa	Continues compilation after preprocessing	2-25
–ppc	Performs preprocessing only. Writes preprocessed output, keeping the comments, to a file with the same name as the input but with a .pp extension	2-25
–ppd	Performs preprocessing only, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility	2-25
–ppi	Performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive	2-25
–ppl	Performs preprocessing only. Writes preprocessed output with line-control information (#line directives) to a file with the same name as the input but with a .pp extension	2-25
–ppo	Performs preprocessing only. Writes preprocessed output to a file with the same name as the input but with a .pp extension	2-25

(h) Options that control diagnostics

Option	Effect	Page
–pdel *num*	Sets the error limit to *num*. The compiler abandons compiling after this number of errors. (The default is 100.)	2-29
–pden	Displays a diagnostic’s identifiers along with its text	2-29
–pdf *outfile*	Writes diagnostics to *outfile* rather than standard error	2-29
–pdr	Issues remarks (nonserious warnings)	2-29
–pds *num*	Suppresses the diagnostic identified by *num*	2-29
–pdse *num*	Categorizes the diagnostic identified by *num* as an error	2-29
–pdsr *num*	Categorizes the diagnostic identified by *num* as a remark	2-29
–pdsw *num*	Categorizes the diagnostic identified by *num* as a warning	2-30
–pdv	Provides verbose diagnostics that display the original source with line-wrap	2-30
–pdw	Suppresses warning diagnostics (errors are still issued)	2-30

Table 2–1. Shell Options Summary (Continued)

(i) 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 or –oL0	Informs the optimizer that your file alters a standard library function	3-4
–ol1 or –oL1	Informs the optimizer that your file declares a standard library function	3-4
–ol2 or –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)

(j) Options that control the assembler

Option	Effect	Page
–aa	Enables absolute listing	2-19
–ac	Makes case insignificant in assembly source files	2-19
–ad *name*	Defines the symbol *name*	2-19
–ahc *fileneame*	Copies filenames	2-19
–ahi *fileneame*	Includes filenames	2-19
–al	Generates an assembly listing file	2-19
–as	Puts labels in the symbol table	2-19
–au *name*	Undefines the predefined constant *name*	2-19
–ax	Generates the cross-reference file	2-19

2.3.1 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.

–abs Generates an absolute listing file. This option must be specified after the –z option. With this option, the shell invokes the absolute lister as described in the *TMS320C28x Assembly Language Tools User's Guide.*

–b Generates an auxiliary information file that you can refer to for information about stack size and function calls. The filename is the C/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 4.3, *Disabling the Linker (–c Shell Option)*, on page 4-5.

–d*name[=def]*	Predefines the constant *name* for the preprocessor. This is equivalent to inserting #define *name def* at the top of each C/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/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 an infinite number of –i options per invocation of the compiler; 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 2.5.2.1, *Changing the #include File Search Path(–i Option)*, on page 2-24.
–k	Retains the assembly language output from the compiler. Normally, the shell deletes the output assembly language file after assembly is complete.
–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.
–pn	Disables intrinsics that are provided by the compiler. Whel compiling with the –pn option, if a call is made to one of these intrinsics, the call is treated as a normal function call.
–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/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/C++ statement. The –s option implies the –k option. The –s option will override a previous use of the –ss option.

–ss	Invokes the interlist utility, which interweaves original C/C++ source with compiler-generated assembly language. This option might reorganize your code substantially. For more information, see section 2.10, *Using the Interlist Utility,* on page 2-39. The –s option will override a previous use of the –ss option.
–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/C++ Code*.

2.3.2 Machine–Specific Options

Following are detailed descriptions of machine-specific options that you will probably use often:

–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.
–md	Disables the compiler from optimizing redundant loads of the DP register when using DP direct addressing.
–me	Prevents the compiler from generating TMS320C28x fast branch instructions (BF). Fast branch instructions are generated by the compiler by default when possible.
–mf	Optimizes your code for speed over size. By default, the C28x optimizer attempts to reduce the size of your code at the expense of speed.
–mi	Prevents the compiler from generating repeat (RPT) instructions. By default, repeat instructions are generated for certain memcpy operations and certain division operations. However, repeat instructions are not interruptible.

–ml	Generates large memory model code. This forces the compiler to view the architecture as having a flat 22-bit address space. All pointers will be considered to be 22-bit pointers. The main use of –ml is with C++ code to access memory beyond 16 bits. For more information, see section 6.7.7, *Using the Large Memory Model (–ml Option)*, on page 6-18.
–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.
–ms	Increases the level of code-size optimization performed by the compiler. For more information, see section 3.8, *Increasing Code-Size Optimizations (–ms Option)*, on page 3-17.
–mt	Use this switch if your memory map is configured as a single unified space. This will allow the compiler to generate RPT PREAD instructions for most memcpy calls and structure assignments. This will also allow MAC instructions to be generated. This switch will also allow more efficient data memory instructions to be used to access switch tables.
–mu	Encodes C2xlp OUT instructions as C28x UOUT instructions. The C28x processor has protected (OUT) and unprotected (UOUT) instructions. By default, the assembler encodes C2xlp OUT instructions as C28x protected OUT instructions. The –mu option is ignored if –m20 is not specified.
–mx	Expands macros in an assembly file and assembles the expanded file. Expanding macros helps you to debug the assembly file. The –mx option affects only the assembly file. When –mx is used, the compiler first invokes the assembler with the –ml option to generate the macro–expanded source .exp file. Then the .exp file is assembled to generate the object file. The debugger uses the .exp file for debugging. The .exp file is an intermediate file and any update to this file will be lost. You need to make any updates to the original assembly file.
–v28	Generates code for the TMS320C28x architecture.

2.3.3 Specifying Filenames

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

Extension	**File Type**
.asm, .abs, or .s* (extension begins with s)	Assembly source
.o*	Object
.C, .cpp, .cxx, or .cc†	C++ source
Anything else	C source

† Case sensitivity in filename extensions is determined by your operating system. If your operating system is not case sensitive, .C is interpreted as a C file.

The conventions for filename extensions allow you to compile C/C++ files and optimize and assemble assembly files with a single command.

For information about how you can alter the way that the shell interprets individual filenames, see section 2.3.4, *Changing How the Shell Program Interprets Filenames (–fa, –fc, –fo, and –fp 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 2.3.6, *Specifying Directories*, on page 2-18.

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:

```
cl2000 –v28 *.cpp
```

2.3.4 Changing How the Shell Program Interprets Filenames (–fa, –fc, –fo, and –fp 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, –fo, and –fp 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
–fp*filename*	for a C++ source 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:

```
cl2000 –v28 –fc file.s –fa assy
```

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

The –fg option causes the compiler to process C files as C++ files. By default, the compiler treats files with a .c extension as C files. See section 2.3.3, Specifying Filenames, on page 2-16, for more information about filename extension conventions.

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

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

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

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

```
cl2000 –v28 –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:

```
cl2000 –v28 –earrr –eoo fit.rrr
```

2.3.6 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:

–fb Specifies the destination directory for absolute listing files. The default is to use the same directory as the object file directory. To specify an absolute listing file directory, type the directory’s pathname on the command line after the –fb option:

```
cl2000 –v28 –fb d:\abso_list
```

–ff Specifies the destination directory for assembly listing files and cross-reference listing files. The default is to use the same directory as the object file directory. To specify an assembly/cross-reference listing file directory, type the directory’s pathname on the command line after the –ff option:

```
cl2000 –v28 –ff d:\listing
```

–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:

```
cl2000 –v28 –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:

```
cl2000 –v28 –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:

```
cl2000 –v28 –ft c:\temp
```

2.3.7 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.
–ac	Makes case insignificant in the assembly source files. For example, –ac makes the symbols ABC and abc equivalent. If you do not use this option, case is significant (this is the default.)
–ad_name_[_value_]	Sets the _name_ symbol. This is equivalent to inserting _name_ .set[_value_] at the beginning of the assembly file. If _value_ is omitted, the symbol is set to 1.
–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.
–au_name_	Undefines the predefined constant _name_, which overrides any –ad options for the specified constant.
–ax	Invokes the assembler with the –x assembler option to produce a symbolic cross-reference in the listing file.

For more information about assembler options, see the *TMS320C28x Assembly Language Tools User's Guide.*

2.4 Changing the Compiler’s Behavior With Environment Variables

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

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

2.4.1 Identifying Alternative Directories for the Compiler to Search (C_DIR)

The compiler uses the C_DIR environment variable to name alternative directories for the compiler to search. To set the C_DIR environment variable, use this syntax:

set C_DIR=*pathname*$_1$ [;*pathname*$_2$ . . .]

The *pathnames* are directories that contain #include files or function libraries (such as stdio.h). You can separate the pathnames with a semicolon or with a blank. In C source, you can use the #include directive without specifying path information. Instead, you can specify the path information with C_DIR.

2.4.2 Setting Default Shell Options (C_OPTION)

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

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

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

Operating System	Enter
UNIX with C shell	**setenv C_OPTION ”***option*$_1$ [*option*$_2$ . . .]**”**
UNIX with Bourne or Korn shell	**C_OPTION=”***option*$_1$ [*option*$_2$ . . .]**”** **export C_OPTION**
Windows	**set C_OPTION=***option*$_1$ [;*option*$_2$ . . .]

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

```
set C_OPTION=–qs –z
```

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

```
cl2000 –v28 *c                ; compiles and links
cl2000 –v28 –c *.c            ; only compiles
cl2000 –v28 *.c –z lnk.cmd    ; compiles and links using
                              ; a command file
cl2000 –v28 –c *.c –z lnk.cmd ; only compiles
                              ; (–c overrides –z)
```

For more information about shell options, see section 2.3, *Changing the Compiler’s Behavior With Options*, on page 2-5. For more information about linker options, see section 4.4, *Linker Options*, on page 4-6*.*

2.4.3 Specifying a Temporary File Directory (TMP)

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

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

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

Operating System	**Enter**
UNIX with C shell	**setenv TMP ”***pathname***”**
UNIX with Bourne or Korn shell	**TMP=”***pathname***”** **export TMP**
Windows	**set TMP=***pathname*

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

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

```
set TMP=c:\temp ↵
```

2.5 Controlling the Preprocessor

This section describes specific features that control the C28x preprocessor, which is part of the parser. A general description of C preprocessing is presented in section A12 of K&R. The C28x C/C++ compiler includes standard C/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.

2.5.1 Predefined Macro Names

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

Table 2–2. Predefined Macro Names

Macro Name	Description
_ _LINE_ _†	Expands to the current line number
_ _FILE_ _†	Expands to the current source filename
_ _DATE_ _†	Expands to the compilation date in the form *mm dd yyyy*
_ _TIME_ _†	Expands to the compilation time in the form *hh:mm:ss*
_ INLINE	Expands to 1 under the –x or –x2 option; undefined otherwise
_ _TMS320C28XX_ _	Expands to 1 (identifies the C28x 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”);
```

2.5.2 The Search Path for #include Files

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

- If you enclose the filename in double quotes (“ ”), the compiler searches for the file in the following directories in 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 2.5.2.1, *Changing the #include File Search Path (–i Option*) for information on using the –i option.

2.5.2.1 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/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:\c28xtools\files\alt.h
UNIX	/c28xtools/files/alt.h

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

Operating System	Enter
Windows	`cl2000 –v28 –ic:\c28xtools\files source.c`
UNIX	`cl2000 –v28 –i/c28xtools/files source.c`

Note: Specifying Path Information in Angle Brackets

If you specify the path information in angle brackets, the compiler applies that information *relative* to the path information specified with –i options and the C_DIR environment variable.

For example, if you set up C_DIR with the following command:

```
set C_DIR= c:\project\include
```

or invoke the compiler with the following command:

```
cl2000 –v28 –i c:\project\include file.c
```

and file.c contains this line:

```
#include <module\proc.h>
```

the result is that the included file is in the following path:

```
c:\project\include\module\proc.h
```

2.5.3 Generating a Preprocessed Listing File (–ppo Option)

The –ppo 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.
- ❑ 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**.**

2.5.4 Continuing Compilation After Preprocessing (–ppa Option)

If you are preprocessing, the preprocessor performs preprocessing only—by default, it does not compile your source code. If you want to override this feature and continue to compile after your source code is preprocessed, use the –ppa option along with the other preprocessing options. For example, use –ppa with –ppo to perform preprocessing, write preprocessed output to a file with a .pp extension, and then compile your source code.

2.5.5 Generating a Preprocessed Listing File With Comments (–ppc Option)

The –ppc option performs all of the preprocessing functions except removing comments and generates a preprocessed version of your source file with a .pp extension. Use the –ppc option instead of the –ppo option if you want to keep the comments.

2.5.6 Generating a Preprocessed Listing File With Line-Control Information (–ppl Option)

By default, the preprocessed output file contains no preprocessor directives. If you want to include the #line directives, use the –ppl option. The –ppl option performs preprocessing only and writes preprocessed output with line-control information (#line directives) to a file with the same name as the source file but with a .pp extension.

2.5.7 Generating Preprocessed Output for a Make Utility (–ppd Option)

The –ppd option performs preprocessing only, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility. The list is written to a file with the same name as the source file but with a .pp extension.

2.5.8 Generating a List of Files Included With the #include Directive (–ppi Option)

The –ppi option performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive. The list is written to a file with the same name as the source file but with a .pp extension.

2.6 Understanding Diagnostic Messages

One of the compiler's primary functions is to report diagnostics for the source program. When the compiler detects a suspect condition, it displays a message in the following format:

"*file.cpp***", line** *n***:** *diagnostic severity***:** *diagnostic message*

"*file.cpp***"**	The name of the file involved
line *n***:**	The line number where the diagnostic applies
diagnostic severity	The severity of the diagnostic message (a description of each severity category follows)
diagnostic message	The text that describes the problem

Diagnostic messages have an associated severity, as follows:

- ❑ A **fatal error** indicates a problem of such severity that the compilation cannot continue. Examples of problems that can cause a fatal error include command-line errors, internal errors, and missing include files. If multiple source files are being compiled, any source files after the current one will not be compiled.
- ❑ An **error** indicates a violation of the syntax or semantic rules of the C or C++ language. Compilation continues, but object code is not generated.
- ❑ A **warning** indicates something that is valid but questionable. Compilation continues and object code is generated (if no errors are detected).
- ❑ A **remark** is less serious than a warning. It indicates something that is valid and probably intended, but may need to be checked. Compilation continues and object code is generated (if no errors are detected). By default, remarks are not issued. Use the –pdr shell option to enable remarks.

Diagnostics are written to standard error with a form like the following example:

```
"test.c", line 5: error: a break statement may only be used
          within a loop or switch
    break;
    ^
```

By default, the source line is omitted. Use the –pdv shell option to enable the display of the source line and the error position. The above example makes use of this option.

The message identifies the file and line involved in the diagnostic, and the source line itself (with the position indicated by the ^ symbol) follows the message. If several diagnostics apply to one source line, each diagnostic has the form shown; the text of the source line is displayed several times, with an appropriate position indicated each time.

Long messages are wrapped to additional lines, when necessary.

You can use a command-line option (–pden) to request that the diagnostic's numeric identifier be included in the diagnostic message. When displayed, the diagnostic identifier also indicates whether the diagnostic can have its severity overridden on the command line. If the severity can be overridden, the diagnostic identifier includes the suffix –D (for *discretionary*); otherwise, no suffix is present. For example:

```
”Test_name.c”, line 7: error #64–D: declaration does not
          declare anything
    struct {};
    ^

”Test_name.c”, line 9: error #77: this declaration has no
          storage class or type specifier
    xxxxx;
    ^
```

Because an error is determined to be discretionary based on the error severity associated with a specific context, an error can be discretionary in some cases and not in others. All warnings and remarks are discretionary.

For some messages, a list of entities (functions, local variables, source files, etc.) is useful; the entities are listed following the initial error message:

```
”test.c”, line 4: error: more than one instance of overloaded
          function ”f” matches the argument list:
            function ”f(int)”
            function ”f(float)”
            argument types are: (double)
    f(1.5);
    ^
```

In some cases, additional context information is provided. Specifically, the context information is useful when the front end issues a diagnostic while doing a template instantiation or while generating a constructor, destructor, or assignment operator function. For example:

```
”test.c”, line 7: error: ”A::A()” is inaccessible
    B x;
       ^
          detected during implicit generation of ”B::B()” at
          line 7
```

Without the context information, it is difficult to determine to what the error refers.

2.6.1 Controlling Diagnostics

The C/C++ compiler provides diagnostic options that allow you to modify how the parser interprets your code. You can use these options to control diagnostics:

–pdel *num*	Sets the error limit to *num*, which can be any decimal value. The compiler abandons compiling after this number of errors. (The default is 100.)
–pden	Displays a diagnostic's numeric identifier along with its text. Use this option in determining which arguments you need to supply to the diagnostic suppression options (–pds, –pdse, –pdsr, and –pdsw). This option also indicates whether a diagnostic is discretionary. A discretionary diagnostic is one whose severity can be overridden. A discretionary diagnostic includes the suffix –D; otherwise, no suffix is present. See section 2.6, *Understanding Diagnostic Messages*, for more information.
–pdf *outfile*	Writes diagnostics to *outfile* rather than to standard error
–pdr	Issues remarks (nonserious warnings), which are suppressed by default
–pds *num*	Suppresses the diagnostic identified by *num*. To determine the numeric identifier of a diagnostic message, use the –pden option first in a separate compile. Then use –pds *num* to suppress the diagnostic. You can suppress only discretionary diagnostics.
–pdse *num*	Categorizes the diagnostic identified by *num* as an error. To determine the numeric identifier of a diagnostic message, use the –pden option first in a separate compile. Then use –pdse *num* to recategorize the diagnostic as an error. You can alter the severity of discretionary diagnostics only.
–pdsr *num*	Categorizes the diagnostic identified by *num* as a remark. To determine the numeric identifier of a diagnostic message, use the –pden option first in a separate compile. Then use –pdsr *num* to recategorize the diagnostic as a remark. You can alter the severity of discretionary diagnostics only.

–pdsw *num*	Categorizes the diagnostic identified by *num* as a warning. To determine the numeric identifier of a diagnostic message, use the –pden option first in a separate compile. Then use –pdsw *num* to recategorize the diagnostic as a warning. You can alter the severity of discretionary diagnostics only.
–pdv	Provides verbose diagnostics that display the original source with line-wrap and indicate the position of the error in the source line
–pdw	Suppresses warning diagnostics (errors are still issued)

2.6.2 How You Can Use Diagnostic Suppression Options

The following example demonstrates how you can control diagnostic messages issued by the compiler.

Consider the following code segment:

```
int one();
int i;
int main()
{
  switch (i){
  case  1:
        return one ();
        break;
  default:
        return 0;
        break;
  }
}
```

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

```
”err.cpp”, line 9: warning: statement is unreachable
”err.cpp”, line 12: warning: statement is unreachable
```

Because it is standard programming practice to include break statements at the end of each case arm to avoid the fall-through condition, these warnings can be ignored. Using the –pden option, you can find out the diagnostic identifier for these warnings. Here is the result:

```
[err.cpp]
”err.cpp”, line 9: warning #111–D: statement is unreachable
”err.cpp”, line 12: warning #111–D: statement is unreachable
```

Next, you can use the diagnostic identifier of 111 as the argument to the –pdsr option to treat this warning as a remark. This compilation now produces no diagnostic messages (because remarks are disabled by default).

Although this type of control is useful, it can also be extremely dangerous. The compiler often emits messages that indicate a less than obvious problem. Be careful to analyze all diagnostics emitted before using the suppression options.

2.6.3 Other Messages

Other error messages that are unrelated to the source, such as incorrect command-line syntax or inability to find specified files, are usually fatal. They are identified by the symbol >> preceding the message.

2.7 Generating Cross-Reference Listing Information (–px *Option*)

The –px shell option generates a cross-reference listing file that contains reference information for each identifier in the source file. (The –px option is separate from –ax, which is an assembler rather than a parser option.) The information in the cross-reference listing file is displayed in the following format:

sym-id   name   X   filename   line number   column number

sym-id	An integer uniquely assigned to each identifier
name	The identifier name
X	One of the following values:

X Value	Meaning
D	Definition
d	Declaration (not a definition)
M	Modification
A	Address taken
U	Used
C	Changed (used and modified in a single operation)
R	Any other kind of reference
E	Error; reference is indeterminate

filename	The source file
line number	The line number in the source file
column number	The column number in the source file

2.8 Generating a Raw Listing File (–pl Option)

The –pl option generates a raw listing file that can help you understand how the compiler is preprocessing your source file. Whereas the preprocessed listing file (generated with the –ppo, –ppc, –ppl, and –ppf preprocessor options) shows a preprocessed version of your source file, a raw listing file provides a comparison between the original source line and the preprocessed output. The raw listing file contains the following information:

- ❑ Each original source line
- ❑ Transitions into and out of include files
- ❑ Diagnostics
- ❑ Preprocessed source line if nontrivial processing was performed (comment removal is considered trivial; other preprocessing is nontrivial)

Each source line in the raw listing file begins with one of the identifiers listed in Table 2–3.

Table 2–3. Raw Listing File Identifiers

Identifier	Definition
N	Normal line of source
X	Expanded line of source. It appears immediately following the normal line of source if nontrivial preprocessing occurs.
S	Skipped source line (false #if clause)
L	Change in source position, given in the following format: L *line number filename key* Where *line number* is the line number in the source file. The *key* is present only when the change is due to entry/exit of an include file. Possible values of *key* are as follows: 1 = entry into an include file 2 = exit from an include file

The –pl option also includes diagnostic identifiers as defined in Table 2–4.

Table 2–4. Raw Listing File Diagnostic Identifiers

Diagnostic identifier	Definition
E	Error
F	Fatal
R	Remark
W	Warning

Diagnostic raw listing information is displayed in the following format:

S   filename   line number   column number   diagnostic

S	One of the identifiers in Table 2–4 that indicates the severity of the diagnostic
filename	The source file
line number	The line number in the source file
column number	The column number in the source file
diagnostic	The message text for the diagnostic

Diagnostics after the end of file are indicated as the last line of the file with a column number of 0. When diagnostic message text requires more than one line, each subsequent line contains the same file, line, and column information but uses a lowercase version of the diagnostic identifier. For more information about diagnostic messages, see section 2.6, *Understanding Diagnostic Messages*.

2.9 Using Inline Function Expansion

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

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

There are several types of inline function expansion:

- Inlining with intrinsic operators (intrinsics are always inlined)
- Automatic inlining
- Definition-controlled inlining with the unguarded inline keyword
- Definition-controlled inlining with the guarded inline keyword

Note: Function Inlining Can Greatly Increase Code Size

Expanding functions inline increases code size, especially inlining a function that is called in a number of places. Function inlining is optimal for functions that are called only from a small number of places and for small functions.

2.9.1 Inlining Intrinsic Operators

There are many intrinsic operators for the TMS320C28x. All of them are automatically inlined by the compiler. The inlining happens automatically whether or not you use the optimizer.

For details about intrinsics, and a list of the intrinsics, see section 7.4.5, *Using Intrinsics to Access Assembly Language Statements*, on page 7-24.

2.9.2 Automatic Inlining

When compiling C/C++ source code with the –o3 option, inline function expansion is performed on small functions . For more information, see section 3.5, *Automatic Inline Expansion (–oi Option)*, on page 3-12.

2.9.3 Unguarded Definition-Controlled Inlining

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

You must invoke the optimizer with any –o option (–o0, –o1, –o2, or –o3) to turn on definition-controlled inlining. Automatic inlining is also turned on when using –o3.

The following example shows usage of the inline keyword, where the function call is replaced by the code in the called function.

Example 2–1. Using the inline keyword

```
inline int volume_sphere(float r)
{
   return 4.0/3.0 * PI * r * r * r;
}
int foo(...)
{
   ...
   volume = volume_sphere(radius);
   ...
}
```

The –pi option turns off definition-controlled inlining. This option is useful when you need a certain level of optimization but do not want definition-controlled inlining.

2.9.4 Guarded Inlining and the _INLINE Preprocessor Symbol

When declaring a function in a header file as static inline, additional procedures should be followed to avoid a potential code size increase when inlining is turned off with –pi or the optimizer is not run.

In order to prevent a static inline function in a header file from causing an increase in code size when inlining gets turned off, use the following procedure. This allows external-linkage when inlining is turned off; thus, only one function definition will exist throughout the object files.

- ❑ Prototype a static inline version of the function. Then, prototype an alternative, nonstatic, externally-linked version of the function. Conditionally preprocess these two prototypes with the _INLINE preprocessor symbol, as shown in Example 2–2.
- ❑ Create an identical version of the function definition in a .c or .cpp file, as shown in Example 2–2.

In Example 2–2 there are two definitions of a 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 (_INLINE is automatically defined for you when the optimizer is used and –pi is not specified).

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

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

(a) foo.h

```
#ifdef _INLINE
#define _IDECL static inline
#else
#define _IDECL extern
#endif
_IDECL int len (const char *str);
#ifdef _INLINE
static inline int len(const char *str)
{
   int n         = –1;
   const char *s = str – 1;
   do n++; while (*++s);
   return n;
}

#endif
```

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

(b) foo.c

```
#undef _INLINE

#include ”foo.h”

int len(cont char * str)
{
   int n         = –1;
   const char *s = str – 1;

   do n++; while (*++s);
   return n;
}
```

2.9.5 Inlining Restrictions

There are several restrictions on what functions can be inlined for both automatic inlining and definition-controlled inlining. Functions with local static variables or a variable number of arguments are not inlined, with the exception of functions decalred as static inline. In functions declared as static inline, expansion occurs despite the presence of local static variables. In addition, a limit is placed on the depth of inlining for recursive or nonleaf functions. Furthermore, inlining should be used for small functions or functions that are called in a few places (though the compiler does not enforce this).

A function may be disqualified from inlining if it:

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

2.10 Using the Interlist Utility

The compiler tools include a utility that interlists C/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/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:

```
cl2000 –v28 –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/C++ source files, merges them, and writes the C/C++ statements into the assembly file as comments.

Example 2–3 shows a typical interlisted assembly file.

Example 2–3. An Interlisted Assembly Language File

```
;----------------------------------------------------------------------
;   1 | int main()
;----------------------------------------------------------------------

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

_main:
;----------------------------------------------------------------------
;   3 | printf(”Hello World\n”);
;----------------------------------------------------------------------
        MOVL      XAR4,#SL1             ; |3|
        LCR       #_printf              ; |3|
        ; call occurs [#_printf] ; |3|
;----------------------------------------------------------------------
;   4 | return 0;
;----------------------------------------------------------------------

;***************************************************************
;* 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.

Chapter 3

Optimizing Your Code

The compiler tools include an optimization program that improves the execution speed and reduces the size of C/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**

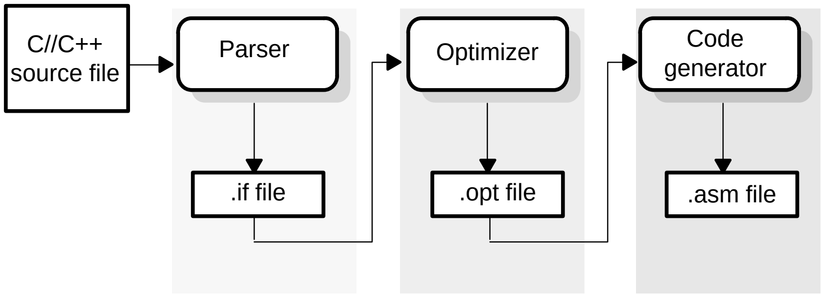
3.1 Using the C/C++ Compiler Optimizer

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 cl2000 –v28 shell program, specifying the –o option.

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 (g) *Options that control the optimizer*, on page 2-11.

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

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



To invoke the optimizer, use the cl2000 –v28 shell program, specifying the –o option 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
  - Reorders function declarations so that the attributes of called functions are known when the caller is optimized
  - Propagates arguments into function bodies when all call sites pass the same value in the same argument position
  - Identifies file-level variable characteristics

  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 C28x-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 Performing File-Level Optimization

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

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

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

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

When you invoke the optimizer with the –o3 option, some of the optimizations use known properties of the standard library functions. If your file redeclares any of these standard library functions, these optimizations become ineffective. The –ol 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 if functions in other modules can call a module's external functions or modify a module's external variables. The number following –op indicates the level you set for the module that you are allowing to be called or modified. The –o3 option combines this information with its own file-level analysis to decide whether to treat this module's external function and variable declarations as if they had been declared static. Use Table 3–4 to select the appropriate level to append to the –op option.

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

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

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

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

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

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

3.3.2 Optimization Considerations When Mixing C/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/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/C++ functions, the –pm option optimizes out those C/C++ functions.

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.

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/C++ source code that calls assembly functions. Those assembly functions do not call any C/C++ functions or modify any C/C++ variables.

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

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

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

Situation

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

Solution

You *must* add the volatile keyword to the C/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.

3.4 Special Considerations When Using the Optimizer

The optimizer is designed to improve your ANSI-conforming C and 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/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/C++ environment or access C/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/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 (–oi Option)

When optimizing with the –o3 option, the compiler automatically inlines small functions. A command-line option, –oi*size*, specifies the size threshold. Any function larger than the *size* threshold is not automatically inlined. You can use the –oi*size* option in the following ways:

- ❑ If you set the *size* parameter to 0 (–oi0), automatic inline expansion is disabled.
- ❑ If you set the *size* parameter to a nonzero integer, the compiler uses this size threshold as a limit to the size of the functions it automatically inlines. The optimizer multiplies the number of times the function is inlined (plus 1 if the function is externally visible and its declaration cannot be safely removed) by the size of the function.

The compiler inlines the function only if the result is less than the size parameter. The compiler measures the size of a function in arbitrary units; however, the optimizer information file (created with the –on1 or –on2 option) reports the size of each function in the same units that the –oi option uses.

The –oi*size* option controls only the inlining of functions that are not explicitly declared as inline. If you do not use the –oi*size* option, the optimizer inlines very small functions. .

Note: The –o3 Option Optimization and Inlining

In order to turn on automatic inlining, you must use the –o3 option. The –o3 option turns on other optimizations. If you desire the –o3 optimizations, but not automatic inlining, use –oi0 with the –o3 option.

Note: Inlining and Code Size

Expanding functions inline increases code size, especially inlining a function that is called in a number of places. Function inlining is optimal for functions that are called only from a small number of places and for small functions. In order to prevent increases in code size because of inlining, use the –oi0 and –pi options.

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/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/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/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/C++ statements.

Example 3–1 shows a function compiled with the optimizer (–o2) and the –os option. The assembly file contains optimizer comments interlisted with assembly code.

Example 3–1. C Code Compiled With the –o2 and –os Options

(a) C source

```
int copy (char *str, const char *s, int n)
{
   int i;

   for (i = 0; i < n; i ++)
      *str++ = *s++;
}
```

Example 3–1.C Code Compiled With the –o2 and –os Options (Continued)

(b) Compiler output

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

_copy:
;*** 6  -----------------------    if ( n <= 0 ) goto g4;
        CMPB      AL,#0                ; |6|
        B         L2,LEQ               ; |6|
        ; branch occurs ; |6|
;***    -----------------------    #pragma MUST_ITERATE(1, 4294967295, 1)
:***    -----------------------    L$1 = n–1;
        ADDB      AL,#–1
        MOVZ      AR6,AL
L1:
;***    -----------------------g3:
;*** 7  -----------------------    *str++ = *s++;
;*** 7  -----------------------    if ( (––L$1) != (–1) ) goto g3;
        MOV       AL,*XAR5++           ; |7|
        MOV       *XAR4++,AL           ; |7|
        BANZ      L1,AR6––
        ; branch occurs ; |7|
;***    -----------------------g4:
;***    -----------------------    return;
L2:
        LRETR
        ; 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/C++ source into the assembly file.

Example 3–2 shows the function from Example 3–1 compiled with the optimizer (–o2) and the –ss and –os options. The assembly file contains optimizer comments and C source interlisted with assembly code.

Example 3–2. Example 3–1(a) Compiled With the –o2, –os, and –ss Options

```
;----------------------------------------------------------------------
;   2 | int copy (char *str, const char *s, int n)
;----------------------------------------------------------------------

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

_copy:
;* AR4   assigned to _str
;* AR5   assigned to _s
;* AL    assigned to _n
;* AL    assigned to _n
;* AR5   assigned to _s
;* AR4   assigned to _str
;* AR6   assigned to L$1
;*** 6  -----------------------    if ( n <= 0 ) goto g4;
;----------------------------------------------------------------------
;   4 | int i;
;----------------------------------------------------------------------
;----------------------------------------------------------------------
;   6 | for (i = 0; i < n; i++)
;----------------------------------------------------------------------
        CMPB      AL,#0                ; |6|
        B         L2,LEQ               ; |6|
        ; branch occurs ; |6|
;***    -----------------------    #pragma MUST_ITERATE(1, 4294967295, 1)
:***    -----------------------    L$1 = n–1;
        ADDB      AL,#–1
        MOVZ      AR6,AL
        NOP
L1:
;***    -----------------------g3:
;*** 7  -----------------------    *str++ = *s++;
;*** 7  -----------------------    if ( (––L$1) != (–1) ) goto g3;
;----------------------------------------------------------------------
;   7 | *str++ = *s++;
;----------------------------------------------------------------------
        MOV       AL,*XAR5++           ; |7|
        MOV       *XAR4++,AL           ; |7|
        BANZ      L1,AR6––
        ; branch occurs ; |7|
;***    -----------------------g4:
;***    -----------------------    return;
L2:
        LRETR
        ; 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/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 Increasing Code-Size Optimizations (–ms Option)

The –ms option increases the level of code-size optimizations performed by the compiler. These optimizations are done at the expense of performance. The optimizations include procedural abstraction where common blocks of code are replaced with function calls. For example, prolog and epilog code, certain intrinsics, and other common code sequences, can be replaced with calls to functions that are defined in the run-time library. It is necessary to link with the supplied run-time library when using the –ms option. It is not necessary to use the optimizer to invoke the –ms option.

To illustrate how the –ms option works, the following describes how prolog and epilog code can be replaced. This code is changed to function calls depending on the number of SOE registers, the size of the frame, and whether a frame pointer is used. These functions are defined in each file with the –ms option, as shown below:

```
_prolog_c28x_1
_prolog_c28x_2
_prolog_c28x_3
_epilog_c28x_1
_epilog_c28x_2
```

Example 3–3 provides an example of C code that was compiled with the –ms option and the resulting output.

Example 3–3. Code Compiled With the –ms Option

(a) C source code

```
extern int x, y, *ptr;
extern int foo();

int main(int a, int b, int c)
{
   ptr[50] = foo();
   y = ptr[50] + x + y + a +b + c;
}
```

(b) Compiler output

```
FP      .set    XAR2
        .global _prolog_c28x_1
        .global _prolog_c28x_2
        .global _prolog_c28x_3
        .global _epilog_c28x_1
        .global _epilog_c28x_2
        .sect   “.text”
        .global _main

;****************************************************************
;* FNAME: _main                        FR SIZE:   6            *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                              0 Parameter,  0 Auto,  6 SOE   *
;****************************************************************

_main:
        FFC       XAR7,_prolog_c28x_1
        MOVZ      AR3,AR4               ; |5|
        MOVZ      AR2,AH                ; |5|
        MOVZ      AR1,AL                ; |5|
        LCR       #_foo                 ; |6|
        ; call occurs [#_foo] ; |6|
        MOVW      DP,#_ptr
        MOVL      XAR6,@_ptr            ; |6|
        MOVB      XAR0,#50              ; |6|
        MOVW      DP,#_y
        MOV       *+XAR6[AR0],AL        ; |6|
        MOV       AH,@_y                ; |7|
        MOVW      DP,#_x
        ADD       AH,AL                 ; |7|
        ADD       AH,@_x                ; |7|
        ADD       AH,AR3                ; |7|
        ADD       AH,AR1                ; |7|
        ADD       AH,AR2                ; |7|
        MOVB      AL,#0
        MOVW      DP,#_y
        MOV       @_y,AH                ; |7|
        FFC       XAR7,_epilog_c28x_1
        LRETR
        ; return occurs
```

3.9 What Kind of Optimization Is Being Performed?

The TMS320C28x C/C++ compiler uses a variety of optimization techniques to improve the execution speed of your C/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/C++ code:

Optimization	Page
Cost-based register allocation	3-20
Alias disambiguation	3-21
Data flow optimizations ❑ Copy propagation ❑ Common subexpression elimination ❑ Redundant assignment elimination	3-21
Expression simplification	3-22
Inline expansion of run-time-support library functions	3-23
Induction variable optimizations and strength reduction	3-25
Loop-invariant code motion	3-25
Loop rotation	3-25
Register variables	3-25
Register tracking/targeting	3-25

3.9.1 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, the induction variable i has been eliminated. This allows the use of the RPT instruction which implements the counting loop. Strength reduction turns the array references into efficient pointer references with auto increments.

Example 3–4. 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;
}
```

(b) Compiler output:

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

_main:
;***    -----------------------    #pragma MUST_ITERATE(10, 10, 10)
;***    -----------------------    U$4 = &a[0];
;***    -----------------------    L$1 = 9;
;***    -----------------------g2:
;*** 7  -----------------------    *U$4++ = 0;
;*** 7  -----------------------    if ( (--L$1) != (-1) ) goto g2;
;*** 7  -----------------------    return 0;|
        MOVL      XAR4,#_a
        MOVB      AL,#0
        RPT       #9
||      MOV       *XAR4++,#0
        LRETR
        ; return occurs
```

3.9.2 Alias Disambiguation

C/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.

3.9.3 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 3–5 on page 3-22).

❑ **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 3–5).

3.9.4 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 3–5).

In Example 3–5, 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 3–5. 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

```
;*****************************************************************
;* 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|
        LRETR
        ; return occurs
```

3.9.5 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 3–6).

In Example 3–6, the compiler finds the code for the C function toupper( ) and replaces the call with the code.

Example 3–6. Inline Function Expansion

(a) C source

```
#include <ctype.h>

char *strupper (char *s)

{
   char *cp;

   for (cp = s; *cp; cp++)
      *cp = toupper (*cp)
   return s;
}
```

Example 3–6.Inline Function Expansion (Continued)

(b) Compiler output

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

_strupper:
;*** 7  -----------------------    cp = s;
;*** 7  -----------------------    goto g4;
        MOVZ      AR6,AR4                ; |7|
        B         L3,UNC                 ; |7|
        ; branch occurs ; |7|
l!:
;***    -----------------------g1:
;*** 144        -----------------------    ch = C$2;  // [0]
;*** 152        -----------------------    if ( (unsigned)ch–97u >25u ) goto g3;
        MOV       AH,AL                  ; |152|
        ADDB      AH,#–97
        CMPB      AH,#25                 ; |152|
        B         L2,HI                  ; |152|
        ; branch occurs ; |152|
;*** 152        -----------------------    ch –= 32;  // [0]
        ADDB      AL,#–32
L2:
;***    -----------------------g3:
;*** 153        -----------------------    *cp++ = ch;  // [0]
        MOV       *XAR6++,AL             ; |153|
L3:
;***    -----------------------g4:
;*** 8  -----------------------    if ( C$2 = *cp ) goto g1;
        MOV       AL,*+XAR6[0]           ; |8|
        BF        L1,NEQ                 ; |8|
        ; branch occurs ; |8|
;*** 9  -----------------------    return s;
        LRETR
        ; return occurs

;* Inlined function references:
;* [  0] toupper
```

3.9.6 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).

3.9.7 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.

3.9.8 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.

3.9.9 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 3–4 on page 3-20).

3.9.10 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 3–7 on page 3-26).

Example 3–7. Register Tracking/Targeting

(a) C source

```
int x, y;

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

(b) Compiler output

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

_main:
;*** 5  -----------------------    x *= 3;
;*** 6  -----------------------    y = x;
;*** 6  -----------------------    return;
        MOVZ      DP,#_x                ; |5|
        MOV       T,@_x                 ; |5|
        MPYB      ACC,T,#3              ; |5|
        MOV       @_x,AL                ; |6|
        LRETR
        ; return occurs
```

3.9.11 Tail Merging

If you are optimizing for code size, tail merging can be very effective for some functions. Tail merging finds basic blocks that end in an identical sequence of instructions and have a common destination. If such a set of blocks is found, the sequence of identical instructions is made into its own block. These instructions are then removed from the set of blocks and replaced with branches to the newly created block. Thus, there is only one copy of the sequence of instructions, rather than one for each block in the set.

In Example 3–8, the addition to a at the end of all three cases is merged into one block. Also, the multiplication by 3 in the first two cases is merged into another block. This results in a reduction of three instructions. In some cases, this optimization adversely affects execution speed by introducing extra branches.

Example 3–8. Tail Merging

(a) C code

```
int func(int a)
{
   if (a < 0)
   {
      a = –a;
      a += f(a)*3;
   }
   else if (a == 0)
   {
      a = g(a);
      a += f(a)*3;
   }
   else
      a += f(a);

   return a;
}
```

Example 3–8.Tail Merging (Continued)

(b) TMS320C28x C/C++ compiler output

```
;***************************************************************
;* FNAME: _func                         FR SIZE:   2           *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                            0 Parameter,  0 Auto,  2 SOE     *
;***************************************************************

_func:
        MOVL      *SP++,XAR2
        CMPB      AL,#0                 ; |3|
        MOVZ      AR2,AL                ; |2|
        B         L2,LT                 ; |3|
        ; branch occurs ; |3|
        CMPB      AL,#0                 ; |8|
        BF        L1,NEQ                ; |8|
        ; branch occurs ; |8|
        LCR       #_g                   ; |10|
        ; call occurs [#_g] ; |10|
        B         L3,UNC                ; |12|
        ; branch occurs ; |12|
L1:
        LCR       #_f                   ; |14|
        ; call occurs [#_f] ; |14|
        MOV       AH,AR2                ; |14|
        ADD       AH,AL                 ; |14|
        MOVZ      AR2,AH                ; |14|
        B         L4,UNC                ; |14|
        ; branch occurs ; |14|
L2:
        NEG       AL                    ; |5|
L3:
        MOVZ      AR2,AL                : |5|
        LCR       #_f                   ; |6|
        ; call occurs [#_f] ; |6|
        MOV       T,AL                  ; |6|
        MPYB      P,T,#3                ; |6|
        MOV       AL,AR2                ; |6|
        ADD       AL,PL                 ; |6|
        MOVZ      AR2,AL                ; |6|
L4:
        MOV       AL,AR2
        MOVL      XAR2,*––SP            ; |16|
        LRETR
        ; return occurs
```

3.9.12 Removing Comparisons to Zero

Because most ALU instructions can modify the status register, explicit comparisons to 0 may be unnecessary. The TMS320C28x C/C++ compiler removes comparisons to 0 if a previous instruction can be modified to set the status register appropriately.

In Example 3–9, the explicit comparison to 0 following the MOV instruction was removed, and MOV set the status register itself.

Example 3–9. Removing Comparisons to Zero

(a) C code

```
int func (int a, int b)
{
    a = b;

    return a < 0 ? a : 1;
}
```

(b) TMS320C28x C/C++ compiler output

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

_func:
;*** 4  -----------------------    return (b < 0) ? b : 1;
        MOV       AL,AH                ; |2|
        B         L1,LT                ; |4|
        ; branch occurs ; |4|
        MOVB      AL,#1                ; |4|
L1:
        LRETR
        ; return occurs
```

Chapter 4

Linking C/C++ Code

The C/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 cl2000 –v28.

This chapter describes how to invoke the linker with each method. It also discusses special requirements of linking C/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 *TMS320C28x 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/C++ programs in a separate step:

lnk2000 {**–c | –cr**} *filenames* [*–options*] [**–o** *name.out*] [**lnk.cmd**]

lnk2000	The command that invokes the linker.
–c | –cr	Options that tell the linker to use special conventions defined by the C/C++ environment. When you use lnk2000, 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.
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.
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 4.4, *Linker Options*.)
–o *name.out*	The –o option names the output file.
lnk.cmd	Contains options, filenames, directives, or commands for the linker.
–l *libraryname*	If you are linking C/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 4.5.1, *Linking With Run-Time-Support Libraries*, on page 4-8.

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/C++ program consisting of modules prog1, prog2, and prog3 (the output file is named prog.out), enter:

```
lnk2000 –c prog1 prog2 prog3 –o prog.out rts2800.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 *TMS320C28x Assembly Language Tools User's Guide*.

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

The options and parameters discussed in this section apply to both methods of linking; however, when you link while compiling, the linker options must follow the –z option (see section 2.2, *Invoking the C Compiler Shell,* on page 2-3).

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:

```
cl2000 –v28 –sq *.c –z c.cmd –o prog.out rts2800.lib
```

First, all of the files in the current directory that have a *.c* extension are compiled using the –s (interlist C/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, rts2800.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

4.3 Disabling the Linker (–c Shell Option)

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

The –c linker option has a different function than and is independent of the –c shell option. By default, the compiler uses the –c linker option when you use the –z option. This tells the linker to use C/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.

4.4 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
–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* words and defines a global symbol that specifies the heap size. The default is 0x400 words.
–farheap *size*	Sets the far heap size (for far dynamic memory allocation) to *size* words and defines a global symbol that specifies the far heap size. The default is 0x400 words.
–i *directory*	Alters the library-search algorithm to look in *directory* before looking in the default location. This option must appear before the –l linker option. The directory must follow operating-system conventions.
–l *filename*	(Lower case L) Controls where the linker looks for the filename

–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/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 *TMS320C28x Assembly Language Tools User's Guide.*

4.5 Controlling the Linking Process

Regardless of the method you choose for invoking the linker, special requirements apply when linking C/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 *TMS320C28x Assembly Language Tools User's Guide.*

4.5.1 Linking With Run-Time-Support Libraries

You must link all C/C++ programs with a run-time-support library. The library contains standard C/C++ functions as well as functions used by the compiler to manage the C/C++ environment. If your runtime-support library is in the current directory, you do not need to use any options to link to it (but you still must include it on the command line). If your runtime-support library is not in the current directory, use the –l option to tell the linker what library name to search for with the –i options and/or C_DIR pathnames. 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:

lnk2000 {**–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 *TMS320C28x 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.

4.5.2 Run-Time Initialization

You must link all C/C++ programs with an object module called *boot.obj*. When a C/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 the appropriate run-time-support library in the link.

The archive libraries listed below contain C/C++ run-time-support function:

- rts2800.lib
- rts2800_ml.lib

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 all global constructors (when using C++)
4) Calls main
5) Calls exit when main returns

Section 8.5, *Summary of Run-Time-Support Functions and Macros*, on page 8-27 describes additional run-time-support functions that are included in the library. These functions include ANSI C standard runtime support.

4.5.3 Global Variable Construction

Global C++ variables having constructors and destructors require their constructors to be called during program initialization and their destructors to be called during program termination. The C/C++ compiler produces a table of constructors to be called at startup. Section 7.8.4, *Initialization Tables*, on page 7-35 discusses the format of this table. The table is contained in a named section called .pinit. The constructors are invoked in the order that they occur in the table. All constructors are called after initialization of global variables (described in section 7.8.2 on page 7-34) and before main( ) is called. Destructors are registered through the atexit( ) system call and therefore are invoked during the call to exit( ).

4.5.4 Specifying the Type of Initialization

The C/C++ compiler produces data tables for autoinitializing global variables. Section 7.8.4, *Initialization Tables*, on page 7-35 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 7.8.5, *Autoinitialization of Variables at Run Time*, on page 7-38).
- ❑ Autoinitializing variables at load time. Global variables are initialized at *load time*. Use the –cr linker option (see section 7.8.6, *Autoinitialization of Variables at Load Time*, on page 7-39).

When you link a C/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 4-4.) 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/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.

4.5.5 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 shows the list of sections created by the compiler and the memory restrictions imposed on them for linking purposes.

Note: Far Linking Issues

Objects that are declared far in C, or large memory model objects are placed in the .ebss/.econst sections, separate from near objects, which are placed in the .bss/.const sections.

Table 4–1. Linking Sections

(a) Initialized sections

Name	Description	Restrictions
.cinit	C initialization records for explicitly initialized global and static variables	Program
.const	Global and static const variables that are explicitly initialized and that are string literals	Low 64K data
.econst	Far constant variables	Anywhere in data
.pinit	Tables for global constructors (C++)	Program
.switch	Tables for implementing switch statements.	Program/Low 64K data (with –mt only)
.text	Executable code and constants	Program

(b) Uninitialized sections

Name	Contents	Restrictions
.bss	Global and static variables	Low 64K data
.ebss	Far global/static variables	Anywhere in data
.stack	Stack space	Low 64K data
.sysmem	Memory for malloc functions	Low 64K data
.esysmem	Memory for far_malloc functions	Anywhere in data

The C/C++ runtime environment supports placing the system heap (.esysmem section) in far memory by providing far_malloc routines. See section 8.5, *Summary of Runtime-Support Functions*, on page 8-27 for a complete description of malloc and far_malloc routines.

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 7.1.1, *Sections*, on page 7-3 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 *TMS320C28x Assembly Language Tools User's Guide*.

4.5.6 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:

`lnk2000 lnk.cmd`

The MEMORY and SECTIONS directives, may require modification to work with your system. See the linker description chapter in the *TMS320C28x 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       */
–o prog.out                 /* Options               */
–m prog.map
–1 rts2800.lib              /* Get run–time support  */

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

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

Chapter 5

Post-Link Optimizer

The TMS320C28x post-link optimizer removes or modifies assembly language instructions to generate better code. The post-link optimizer examines the final addresses of symbols determined by linking and uses this information to make code changes.

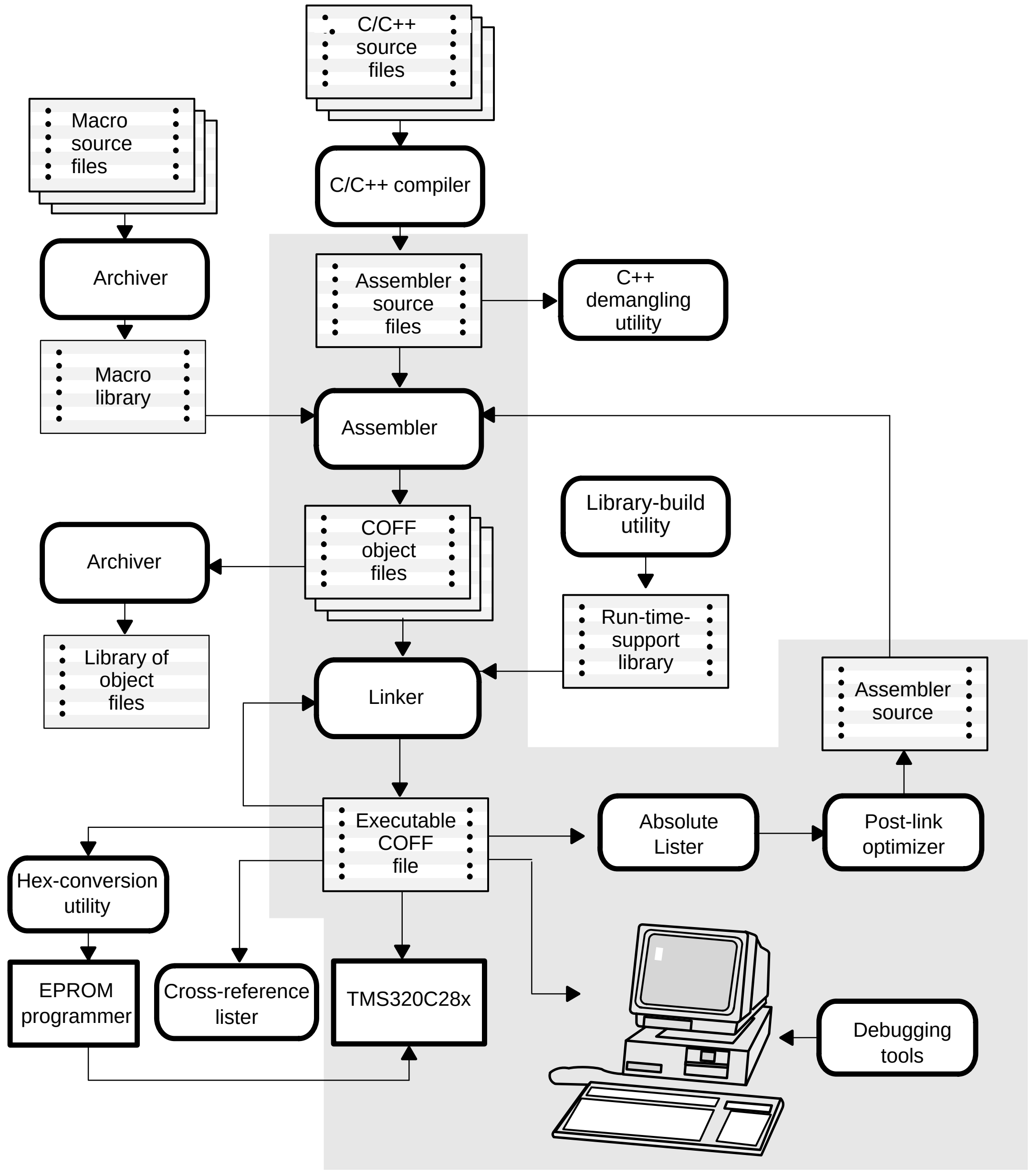
Post-link optimization requires the shell option, –plink. The –plink shell option invokes added passes of the tools that include running the absolute lister (abs2000) and rerunning the assembler and linker. You must use the –plink option following the –z shell option.

Topic **Page**

5.1 The Post-Link Optimizer’s Role in the Software Development Flow

The post-link optimizer is not part of the normal development flow. Figure 5–1 shows the flow including the post-link optimizer; this flow occurs only when you use the shell utility and the –plink option. This flow is shown below:

Figure 5–1. The Post-Link Optimizer in the TMS320C28x Software Development Flow



As the flow shows, the absolute lister (abs2000) is also part of the post-link optimizing process. The absolute lister outputs the absolute addresses of all globally defined symbols and coff sections. The post-link optimizer takes .abs files as input and uses these addresses to perform optimizations. The output is a .pl file, which is an optimized version of the original .asm file. The flow then reruns the assembler and linker to produce a final output file.

The described flow is supported only when you use the shell utility (cl2000 –v28) and the –plink option. If you use a batch file to invoke each tool individually, you must adapt the flow to use the shell utility instead. In addition, you must use the –o option to specify an output file name when using the –plink option. See section 5.8 on page 5-10 for more details.

For example, replace these lines:

```
asm2000 –v28 file1.asm file1.obj
asm2000 –v28 file2.asm file2.obj
lnk2000 file1.obj file2.obj lnk.cmd –o prog.out
```

with this line:

```
cl2000 –v28 file1.asm file2.asm –z lnk.cmd –o prog.out –
plink
```

5.2 Removing Redundant DP Loads

Post-link optimization reduces the difficulty of managing the DP register by removing redundant DP loads. It does this by tracking the current value of the DP and determining whether the address in a MOV DP,#address instruction is located on the same 64-word page to which the DP is currently pointing. If the address can be accessed using the current DP value, the instruction is redundant and can be removed. For example, consider the following code segment:

```
MOVZ      DP,#name1
ADD       @name1,#10
MOVZ      DP,#name2
ADD       @name2,#10
```

If name1 and name2 are linked to the same page, the post-link optimizer determines that loading DP with the address of name2 is not necessary, and it comments out the redundant load.

```
MOVZ      DP,#name1
ADD       @name1,#10
; <<REDUNDANT>>         MOVZ      DP,#name2
ADD       @name2,#10
```

This optimization can be used on C files as well. Even though the compiler manages the DP for all global variable references that are defined within a module, it conservatively emits DP loads for any references to global variables that are externally defined. Using the post-link optimizer can help reduce the number of DP loads in these instances.

5.3 Tracking DP Values Across Branches

In order to track DP values across branches, the post-link optimizer requires that there are no indirect calls or branches, and all possible branch destinations have labels. If an indirect branch or call is encountered, the post-link optimizer will only track the DP value within a basic block. Branch destinations without labels may cause incorrect output from the post-link optimizer.

If the post link optimizer encounters indirect calls or branches, it issues the following warning:

```
NO POST LINK OPTIMIZATION DONE ACROSS BRANCHES
Branch/Call must have labeled destination
```

This warning is issued so that if the file is a hand written assembly file, you can try to change the indirect call/branch to a direct one to obtain the best optimization from the post linker.

5.4 Tracking DP Values Across Function Calls

The post link optimizer optimizes DP loads after a call to a function if the function is defined in the same file scope. For example, consider the following post link optimized code:

```
_main:
    LCR    #_foo
    MOVB    AL, #0
;<<REDUNDANT>>    MOVZ    DP,#_g2
    MOV    @_g2, #20
    LRETR

    .global    _foo
_foo:
    MOVZ    DP, #g1
    MOV      @_g1, #10
    LRETR
```

The MOVZ DP after the function call to _foo is removed by the post link optimizer as the variables _g1 and _g2 are in the same page and the function _foo already set the DP.

In order for the post link optimizer to optimize across the function calls, the functions should have only one return statement. If you are running the post link optimizer on hand written assembly that has more that one return statement per function, the post link optimization output can be incorrect. You can turn off the optimization across function calls by specifying the –nf option after the –plink option.

5.5 Other Post-Link Optimizations

An externally defined symbol used as a constant operand forces the assembler to choose a 16-bit encoding to hold the immediate value. Since the post-link optimizer has access to the externally defined symbol value, it replaces a 16-bit encoding with an 8-bit encoding when possible. For example:

```
.ref ext_sym ; externally defined to be 4
:
:
ADD AL, #ext_sym ; assembly will encode ext_sym with 16
                 ; bits
```

Since ext_sym is externally defined, the assembler chooses a 16-bit encoding for ext_sym. The post-link optimizer changes the encoding of ext_sym to an 8-bit encoding:

```
.ref ext_sym
:
:
; << ADD=>ADDB>> ADD AL,#ext_sym
ADDB AL, #ext_sym
```

Similarly the post link optimizer attempts to reduce the following 2-word instructions to 1-word instructions:

2-Word Instructions	1-Word Instructions
ADD/SUB ACC, *#imm*	ADDB/SUBB ACC, *#imm*
ADD/SUB/AND/OR/XOR/CMP AL, *#imm*	ADDB/SUBB/ANDB/ORB/XORB/CMPB AL, *#imm*
MOVL XAR*n*, *#imm*	MOVB XAR*n*, *#imm*

5.6 Controlling Post-Link Optimizations

There are three ways to control post-link optimizations: by excluding files, by inserting specific comments within an assembly file, and by manually editing the post-link optimization file.

5.6.1 Excluding Files (–ex Option)

Specific files can be excluded from the post-link optimization process by using the –ex option. The files to be excluded must follow the –ex option and include file extensions. The –ex option must be used after the –plink option and no other option may follow. For example:

```
cl2000 –v28 file1.asm file2.asm file3.asm –k –z –lnk.cmd –plink –o prog.out –ex file3.asm
```

The file3.asm will be excluded from the post-link optimization process.

5.6.2 Controlling Post-Link Optimization Within an Assembly File

Within an assembly file, post-link optimizations can be disabled or enabled by using two specially formatted comment statements:

```
;//NOPLINK//
;//PLINK//
```

Assembly statements following the NOPLINK comment are not optimized. Post-link optimization can be reenabled using the //PLINK// comment.

The PLINK and NOPLINK comment format is not case sensitive and there can be white space between the semicolon and PLINK delimiter. The PLINK and NOPLINK comments must appear on a separate line, alone, and must begin in column 1. For example:

```
;      //PLINK//
```

5.6.3 Retaining Post-Link Optimizer Output (–k Option)

The –k option allows you to retain any post-link files (.pl) and .absolute listing files (.abs) generated by the –plink option. Using the –k option lets you view any changes the post-link optimizer makes.

The .pl files contain the commented out statement shown with <<REDUNDANT>> or any improvements to instructions, such as <<ADD=>ADDB>>. The .pl files are assembled and linked again to exclude the commented out lines.

5.6.4 Disable Optimization Across Function Calls (–nf Option)

The –nf option disables the post-link optimization across function calls. The post-link optimizer recognizes the function end by the return statement and assumes there is only one return statement per function. In some hand written assembly code, it is possible to have more than one return statement per function. In such cases, the output of the post-link optimization can be incorrect. You can turn off the optimization across function calls by using the –nf option. This option affects all the files.

5.7 Restrictions on Using the Post-Link Optimizer

The following restrictions affect post-link optimization:

- ❑ Branches or calls to unlabeled destinations invalidate DP load optimizations. All branch destinations must have labels.
- ❑ If the position of the data sections depends on the size of the code sections, the data page layout information used to decide which DP load instructions to remove may no longer be valid.

  For example, consider the following link command file:

  ```
  Sections
  {
      .text    > MEM,
      .mydata  > MEM,
  }
  ```

  A change in the size of the .text section after optimizing causes the .bss section to shift.

  Ensuring that all output data sections are aligned on a 64-word boundary removes this shifting issue.

  For example, consider the following link command file:

  ```
  Sections
  {
      .text > MEM,
      .mydata align = 64 > MEM,
  }
  ```

5.8 Naming the Outfile (–o Option)

When using the –plink option, you must include the –o *filename* option. If the output filename is specified in a linker command file, the shell does not have access to the filename to pass it along to other phases of post-link optimization, and the process will fail. For example:

```
cl2000 –v28 file1.c file2.asm –z –o prog.out lnk.cmd –plink
```

Because the post-link optimization flow uses the absolute lister, abs2000, it must be included in your path.

Chapter 6

TMS320C28x C/C++ Language Implementation

The TMS320C28x ™ C/C++ compiler supports the C language standard developed by a committe of the American National Standards Institute (ANSI) to standardize the C programming language.

The C++ language supported by the TMS320C28x is defined by the ISO/IEC 14882–1998 standard with certain exceptions.

6.1 Characteristics of TMS320C28x 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 TMS320C28x C/C++ compiler.

The following list identifies all such cases and describes the behavior of the TMS320C28x C/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).

6.1.1 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 TMS320C28x 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)

6.1.2 Data Types

The following conventions apply for data types:

- ❑ For information about the representation of data types, see section 6.4, *Data Types*, on page 6-7. (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 long. (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 long. This also true for far pointer subtraction. (ANSI 3.3.6, K&R A7.7)

6.1.3 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)

6.1.4 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)

6.1.5 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 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)

6.1.6 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 (*func*)
- FAST_FUNC_CALL (*func*)

(ANSI 3.8.6, K&R A12.8)

For more information on pragmas, see section 6.8, *Pragma Directives*, on page 6-22.

6.1.7 Header Files

The following applies to header files. For detailed information about header files, see section 8.4, *Header Files*, on page 8-16.

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

6.2 Characteristics of TMS320C28x C++

The TMS320C28x compiler supports C++ as defined in the ISO/IEC 14882:1998 standard. The exceptions to the standard are as follows:

- ❑ Complete C++ standard library support is not included. C subset and basic language support is included.
- ❑ These C++ headers for C library facilities are not included:
  - <ciso646>
  - <clocale>
  - <csignal>
  - <cwchar>
  - <cwctype>
- ❑ These C++ headers are the only C++ standard library header files included:
  - <new>
  - <typeinfo>

  No support for bad_cast or bad_type_id is included in the typeinfo header.
- ❑ Exception handling is not supported.
- ❑ Run-time type information (RTTI) is disabled by default. RTTI can be enabled with the –rtti shell option.
- ❑ The reinterpret_cast type does not allow casting a pointer to member of one class to a pointer to member of a another class if the classes are unrelated.
- ❑ Two-phase name binding in templates, as described in [tesp.res] and [temp.dep] of the standard, is not implemented.
- ❑ Template template parameters are not implemented.
- ❑ The export keyword for templates is not implemented.
- ❑ A typedef of a function type cannot include member function cv-qualifiers.
- ❑ A partial specialization of a class member template cannot be added outside of the class definition.

6.3 Enabling Embedded C++ Mode (–pe Option)

The compiler supports the compilation of embedded C++, a subset of C++, for embedded system programming. The embedded C++ standard omits some features that are of less value or too expensive to support in an embedded system. When compiling for embedded C++, the compiler generates diagnostics for the use of omitted features.

Embedded C++ is enabled by compiling with the –pe shell option.

Embedded C++ omits these C++ features:

- ❑ Templates
- ❑ Exception handling
- ❑ Run-time type information
- ❑ The new cast syntax
- ❑ The keyword *mutable*
- ❑ Multiple inheritance
- ❑ Virtual inheritance

Under the standard definition of embedded C++, namespaces and using-declarations are not supported. The TMS320C28x compiler nevertheless allows these features under embedded C++ because the C++ run-time support library makes use of them. Furthermore, these features impose no run-time penalty.

The TMS320C28x compiler defines the _embedded_cplusplus macro for embedded C++ compile.

The rts.src library supplied with the compiler can be used to link with a module compiled for embedded C++.

6.4 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 6.15, *Limits (float.h and limits.h)*, on page 6-36.

Table 6–1. TMS320C28x 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
far pointers	22 bits	Binary	0	0x3FFFFF

Note: TMS320C28x 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 TMS320C28x 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). TMS320C28x bytes and words are equivalent (16 bits).

6.5 Register Variables

The C/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 7.2, *Register Conventions*, on page 7-11.

6.6 The asm Statement

The TMS320C28x compiler allows you to imbed TMS320C28x assembly language instructions or directives directly into the assembly language output of the compiler. This capability is provided through an extension to the C/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 *TMS320C28x 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/C++ Environment With asm Statements

Be extremely careful not to disrupt the C/C++ environment with asm statements. The compiler does not check the inserted instructions. Inserting jumps and labels into C/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/C++ is unable to access.

6.7 Keywords

The TMS320C28x C/C++ compiler supports the standard const, register, and volatile keywords. In addition, the C28x C/C++ compiler extends the C language through the support of the cregister and interrupt. In C mode, the C/C++ compiler supports the near and far keywords.

6.7.1 The const Keyword

The C/C++ compiler supports the ANSI standard keyword *const*. This keyword gives you greater optimization and control over allocation of storage for certain data objects. You can apply the const qualifier to the definition of any variable or array to ensure that its value is not altered.

The placement of the const keyword within a definition is important. For example, the first statement below defines a constant pointer p to a variable int. The second statement defines a variable pointer q to a constant int:

```
int * const p = &x;
const int * q = &x;
```

6.7.2 The volatile Keyword

The optimizer analyzes data flow to avoid memory accesses whenever possible. If you have code that depends on memory accesses exactly as written in the C code, you must use the volatile keyword to identify these accesses. A variable qualified with a volatile keyword is allocated to an uninitialized section (as opposed to a register). The compiler does 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, define *ctrl as:

```
volatile unsigned int *ctrl;
```

Here the *ctrl pointer is intended to reference a hardware location, such as an interrupt flag.

6.7.3 The cregister Keyword

The compiler extends the C language by adding the cregister keyword to allow high level language access to control registers.

When you use the cregister keyword on an object, the compiler compares the name of the object to a subset of standard control registers for the TMS320C28x (see Table 6–2). If the name matches, the compiler generates the code to reference the control register. If the name does not match, the compiler issues an error.

Table 6–2. Valid Control Registers

Register	Description
IER	Interrupt enable register
IFR	Interrupt flag register

The cregister keyword can only be used in file scope. The cregister keyword is not allowed on any declaration within the boundaries of a function. It can only be used on objects of type integer or pointer. The cregister keyword is not allowed on objects of any floating-point type or on any structure or union objects.

The cregister keyword does not imply that the object is volatile. If the control register being referenced is volatile (that is, can be modified by some external control), then the object must be declared with the volatile keyword also.

To use the control registers in Table 6–2, you must declare each register as follows:

```
extern cregister volatile unsigned int register;
```

Once you have declared the register, you can use the register name directly. Note that IFR is read only and can only be set using the | (or) operation with an immediate. For example:

IFR | = 0x4;

The IFR can only be cleared using the & (AND) operation with an immediate. Any other use of the IFR register will result in an error.

See Example 6–1 for an example that declares and uses control registers.

Example 6–1. Define and Use Control Registers

```
extern cregister volatile unsigned int IFR;
extern cregister volatile unsigned int IER;
extern int x;

main()
{
   IER = x;
   IER |= 0x100;
   printf(”IER = %x\n”, IER);

   IFR &= 0x100;
   IFR |= 0x100;
}
```

6.7.4 The interrupt Keyword

The compiler extends the C/C++ language by adding the interrupt keyword, which specifies that a function is treated as an interrupt function.

Functions that handle interrupts follow special register-saving rules and a special return sequence. When C/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.

6.7.5 The ioport Keyword

The ioport keyword enables access to the I/O port space of the TMS320C28x device. The keyword has the form:

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

ioport	is the keyword that indicates this is a port variable.
type	must be char, short, int, or unsigned int.
port*hex_num*	is the port number, *hex_num* is a hexadecimal number.

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

Example 6–2 shows the use of the ioport keyword. Example 6–2(a) declares the I/O port as the int port 10h, writes a to port 10h and reads prot 10h into b. The example also shows how port variables can be used in other types of expressions.

Example 6–2. Using the ioport Keyword

(a) C source

```
ioport int port10;
int a; int b; int c;
extern void foo(int);

void func()
{
   port10 = a;
   b = port10;
   foo(port10);
   c = port10 + b;
   port10 += a;
}
```

(b) C compiler output

```
_func:
        .line   3
;----------------------------------------------------------------------
;   7 | port10 = a;
;----------------------------------------------------------------------
        MOVZ      DP,#_a
        OUT       *(010H),@_a           ; |7|
        .line   4
;----------------------------------------------------------------------
;   8 | b = port10;
;----------------------------------------------------------------------
        IN        @_b,*(010H)           ; |8|
        .line   5
;----------------------------------------------------------------------
;   9 | foo(port10);
;----------------------------------------------------------------------
        IN        AL,*(010H)            ; |9|
        LCR       #_foo                 ; |9|
        ; call occurs [#_foo] ; |9|
        .line   6
;----------------------------------------------------------------------
;  10 | c = port10 + b;
;----------------------------------------------------------------------
        IN        AL,*(010H)            ; |10|
        MOVZ      DP,#_b
        ADD       AL,@_b                ; |10|
        MOV       @_c,AL                ; |10|
        .line   7
;----------------------------------------------------------------------
;  11 | port10 += a;
;----------------------------------------------------------------------
        IN        AL,*(010H)            ; |11|
        ADD       AL,@_a                ; |11|
        OUT       *(010H),AL            ; |11|
        .line   8
        LRETR
        ; return occurs
```

6.7.6 The far Keyword

The default address space of the C/C++ compiler is limited to the low 64K of memory. All pointers have a default size of 16 bits. The TMS320C28x supports addressing beyond 16 bits. In C, the compiler can access up to four megawords of data by extending the C language with the use of the far type qualifier. A far pointer will have a 22-bit size.

Note: Far support in C++

The TMS320C/C++ compiler does not support the far keyword in C++. Access to far objects in C++ is available through the use of the large memory model option or through intrinsics. For more information, see section 6.7.7, *Using the Large Memory Model (–ml Option)*, on page 6-18; and section 6.7.8, *Using Intrinsics to Access far Memory in C++*, on page 6-19.

6.7.6.1 Semantics

Declaring an object far designates that the object be placed in far memory. This is accomplished by reserving space for that object in a different section than the default .bss. Global variables declared far are placed in a section called .ebss. This section can then be linked anywhere in the TMS320C28x data address space. const objects declared far are similarly placed in the .econst section.

Pointers that are declared to point at far objects have a size of 22 bits. They require two memory locations to store and require the XAR registers to perform addressing.

Note: Pointer Distinction

There is a distinction between declaring a pointer that points at far data (far int *pf) and declaring the pointer itself as far (int*far fp;).

Only global and static variables can be declared far. Nonstatic variables defined in a function (automatic storage class) cannot be far because these variables are allocated on the stack. The compiler will issue a warning and treat these variables as near.

It is meaningless to declare structure members as far. A structure declared far implies that all of its members are far.

6.7.6.2 Syntax

The compiler recognizes far or _ _far as synonymous keywords. The far keyword acts as a type qualifier. When declaring variables, far is used similarly to the type qualifiers const and volatile. Example 6–3 shows the correct way to declare variables.

Example 6–3. Declaring Variables

```
int far sym;        // sym is located in far memory.
far int sym;        // sym is located in far memory.

struct S far s1;    // Likewise for structure s1.

int far *ptr;       // This declares a pointer that points to a far int.
                    // The variable ptr is itself near.

int * far ptr;      // This declares a pointer to a near int. The variable
                    // ptr, however, is located in far memory.

int far * far ptr;  // The pointer and the object it points to are both far.

int far *func();    // Function that returns a pointer to a far int.

int far *memcpy_ff(far void *dest, const far void *src, int count);

                    // Function that takes two far pointers as arguments
                    // and returns a far pointer.

int *far func()     // ERROR: Declares the function as far. Since the
                    // program address space is flat 22–bit, this has no
                    // meaning. Far applies to data only.

int func()
{
int far x;          // ERROR: Far only applies to global/static variables.
:                   // Auto variables on the stack are required to be near
:
int far *ptr        // Ok, since the pointer is on the stack, but points
                    // to far
}


struct S            // Declaring structure members as far is meaningless,
{                   // unless it’s a pointer to far.  Structure objects
int a;              // can, of course, be declared far.
int far b;
int far *ptr;
};
```

6.7.6.3 *Far Run-Time Library Support*

The runtime library has been extended to include far versions of most RTS functions that have pointers as arguments, return values, or that reference RTS global variables. There is also support for managing a far heap. Far RTS support does not include the C I/O routines or any functions that reference C I/O routines. For more information about far RTS support, see section 8.5, *Summary of Runtime-Support Functions*, on page 8-27.

6.7.6.4 *Far Pointer Math*

The ANSI standard states that valid pointer operations are assignment of pointers of the same type, adding or subtracting a pointer and an integer, subtracting or comparing two pointers to members of the same array, and assigning or comparing to zero. All other pointer arithmetic is illegal.

These rules apply to far pointers. The result of subtracting 2 far pointers is a 16-bit integer type. This implies that the compiler does not support pointer subtraction for arrays larger than 64k words in size.

6.7.6.5 *Far String Constants*

For information about placing string constants in far memory, see section 7.1.8, *Character String Constants*, on page 7-9, and section 7.1.9, *far Character String Constants*, on page 7-10.

6.7.6.6 *Allocating .econst to Program Memory*

For information about how to copy the .econst section from program to data memory at boot time, see section 7.1.3, Allocating .const/.econst to Program Memory, on page 7-6.

6.7.7 Using the Large Memory Model (–ml Option)

Since there is no support for the far keyword in C++ code, the large memory model option is provided. The –ml option forces the compiler to view the TMS320C28x architecture as having a flat 22-bit address space. When you compile with the –ml option, all pointers are considered to be 22-bit pointers. There is no 64K word limit on a data type size.

6.7.7.1 Large Memory Model Options

The large memory model option for the compiler shell is –ml. If each compiler tool is invoked separately, the following options must be used:

- ❑ Parser –ml option
- ❑ Optimizer –m option
- ❑ Code Generator –l option
- ❑ Assembler –mf option

The assembler –mf option is used to allow conditional compilation of 16-bit code with large memory model code. The LARGE_MODEL symbol is predefined by the assembler and automatically set to false unless the –mf option is used.

6.7.7.2 Large Memory Model Run-Time-Support Library

The run-time-support libraries support large memory model through condtional compilation. When compiling the run-time-support libraries, the LARGE_MODEL symbol must be defined. This symbol is needed if any of the run-time-support functions that pass a size_t argument or return a size_t argument are accessed in your code. This symbol is also needed if the run-time-support va_arg or offsetof( ) macro is used. Therefore, you should use the –d shell option (see page 2-13) to predefine the LARGE_MODEL symbol when compiling under the large memory model.

6.7.8 Using Intrinsics to Access far Memory in C++

The far keyword extends the C language only. There is no support for the far keyword in C++. Intrinsics are provided to allow access to far memory in C++, along with heap management support routines in the C++ rts library if the large memory model is not used. The intrinsics accept a long integer type that represents an address. The return value of the intrinsic is an implicit far pointer that can be dereferenced to provide access to these basic data types: word, long, and float.

- ❑ _ _farptr_to_word (long address)
- ❑ _ _farptr_to_long (long address)
- ❑ _ _farptr_to_float (long address)

There are two methods used for generating long addresses that can access far memory:

❑ You can use the C++ run-time-support library heap management functions which are provided in the C++ run-time-support library:

  - long far_calloc (unsigned long num, unsigned long size)
  - long far_malloc (unsigned long size)
  - long far_realloc (long ptr, unsigned long size)
  - void far_free (long ptr)

  These functions will allocate memory in the far heap and the intrinsics can then be used to access that memory. For example:

```
#include <stdlib.h>

extern int x;
extern long y;
extern float z;

extern void func1 (int a);
extern void func2 (long b);
extern void func3 (float c);

//create a far object on the heap
long farint = far_malloc (sizeof (int))
long farlong = far_malloc (sizeof (long));
long farfloat = far_malloc (sizeof (float));


//assign a value to the far object
*__farptr_to_word (farint) = 1;
*__farptr_to_word (farint) = x;

*__farptr_to_long (farlong) = 78934;
*__farptr_to_long (farlong) = y;

*__farptr_to_float (farfloat) = 4.56;
*__farptr_to_float (farfloat) = z;


//use far object in expression
x = *__farprt_to_word(farint) + x;
y = *__farptr_to_long(farlong) + y;
z = *__farptr_to_float(farfloat) + z;


//use as argument to function
func1 (*__farptr_to_word (farint));
func2 (*__farptr_to_long (farlong));
func3 (*__farptr_to_float (farfloat));


//free the far object
far_free (farint);
far_free (farlong);
far_free (farfloat);
```

❑ The DATA_SECTION pragma can be used along with inline assembly to place variables in a data section that is linked in far memory. The in-line assembly is used to create a long address to those variables. The intrinsics can then be used to access those variables. For example:

```
#pragma DATA_SECTION (var, “.mydata”)
int var;
extern const long var_addr;
asm (”\t .sect .const”);
asm (”var_addr .long var”);
int x;
x = *__farptr_to_word (var_addr);
```

6.8 Pragma Directives

Pragma directives tell the compiler's preprocessor how to treat functions. The TMS320C28x C/C++ compiler supports the following pragmas:

- CODE_SECTION
- DATA_SECTION
- FAST_FUNC_CALL
- FUNC_EXT_CALLED
- INTERRUPT

The arguments *func* and *symbol* cannot be 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.

Pragma syntax differs between C and C++. In C, you must supply the name of the object or function to which you are applying the pragma as the first argument. In C++, the name is omitted; the pragma applies to the declaration of the object or function that follows it.

6.8.1 The CODE_SECTION Pragma

The CODE_SECTION pragma allocates space for the *func* in a section named *section name*. The CODE_SECTION pragma is useful if you have code objects that you want to link into an area separate from the .text section.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

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

Example 6–4 demonstrates the use of the CODE_SECTION pragma.

Example 6–4. 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];
}
```

(b) Assembly source file

```
        .sect   “.text”
        .global _main

;***************************************************************
;* FNAME: _main                        FR SIZE:   2           *
;*                                                            *
;* FUNCTION ENVIRONMENT                                       *
;*                                                            *
;* FUNCTION PROPERTIES                                        *
;*                            0 Parameter,  1 Auto,  0 SOE     *
;***************************************************************

:_main:
        ADDB      SP,#2
        MOVB      AL,#1                 ; |12|
        LCR       #_funcA               ; |12|
        ; call occurs [#_funcA] ; |12|
        MOV       *–SP[1],AL            ; |12|
        MOVB      AL,#1                 ; |13|
        LCR       #_funcB               ; |13|
        ; call occurs [#_funcB] ; |13|
        MOV       *–SP[1],AL            ; |13|
        SUBB      SP,#2
        LRETR
        ; return occurs
```

Example 6–4. Using the CODE_SECTION Pragma (Continued)

```
        .sect   ”codeA”
        .global _funcA
;***************************************************************
;* FNAME: _funcA                        FR SIZE:   1           *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                            0 Parameter,  1 Auto,  0 SOE     *
;***************************************************************

_funcA:
        ADDB      SP,#1
        MOV       *–SP[1],AL            ; |17|
        MOVZ      AR6,*–SP[1]           ; |18|
        ADD       AR6,#_bufferA         ; |18|
        SUBB      SP,#1                 ; |18|
        MOV       AL,*+XAR6[0]          ; |18|
        LRETR
        ;return occurs

         .sect   “.text”
         .global _funcB

;***************************************************************
;* FNAME: _funcB                        FR SIZE:   1           *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                            0 Parameter,  1 Auto,  0 SOE     *
;***************************************************************

_funcB:
        ADDB      SP,#1
        MOV       *–SP[1],AL            ; |22|
        MOVZ      AR6,*–SP[1]           ; |23|
        ADD       AR6,#_bufferB         ; |23|
        SUBB      SP,#1                 ; |23|
        MOV       AL,*+XAR6[0]          ; |23|
        LRETR
        ;return occurs
```

6.8.2 The DATA_SECTION Pragma

The DATA_SECTION pragma allocates space for the *symbol* in a section called *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 6–5 demonstrates the use of the DATA_SECTION pragma.

The syntax for the pragma in C is:

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

The syntax for the pragma in C++ is:

#pragma DATA_SECTION (“*section name”***)**

Example 6–5. 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
```

6.8.3 The FAST_FUNC_CALL Pragma

The FAST_FUNC_CALL pragma, when applied to a function, generates a TMS320C28x FFC instruction to call the function instead of the CALL instruction. Refer to the *TMS320C28x DSP CPU and Instruction Set User's Guide* for more details on the FFC instruction.

The syntax for the pragma in C is:

#pragma FAST_FUNC_CALL (*func***)**

The syntax for the pragma in C++ is:

#pragma FAST_FUNC_CALL

The FAST_FUNC_CALL pragma should be applied only to a call to an assembly function that returns with the LB *XAR7 instruction. See section 7.4.1, *Using Assembly Language Modules With C/C++ Code*, on page 7-18, for information on combining C/C++ and assembly code.

Since this pragma should be applied only to assembly functions, if the compiler finds a definition for *func* in the file scope, it issues a warning and ignores the pragma.

The following example demonstrates the use of the FAST_FUNC_CALL pragma.

Example 6–6. Using the FAST_FUNC_CALL Pragma

(a) Assembly function

```
_add_long:
        ADD ACC, *–SP[2]
        LB *XAR7
```

(b) C source file

```
#pragma FAST_FUNC_CALL (add_long);

long add_long(long, long);

void foo()
{
   long x, y;
   x = 0xffff;
   y = 0xff;
   y = add_long(x, y);
}
```

(c) Resulting assembly file

```
;***************************************************************
;* FNAME: _foo                         FR SIZE:   6           *
;*                                                            *
;* FUNCTION ENVIRONMENT                                       *
;*                                                            *
;* FUNCTION PROPERTIES                                        *
;*                           2 Parameter,  4 Auto,  0 SOE     *
;***************************************************************

_foo:
        ADDB      SP,#6
        MOVB      ACC,#255
        MOVL      XAR6,#65535           ; |8|
        MOVL      *–SP[6],ACC
        MOVL      *–SP[2],ACC           ; |10|
        MOVL      *–SP[4],XAR6          ; |8|
        MOVL      ACC,*–SP[4]           ; |10|
        FFC       XAR7,#_add_long       ; |10|
        ; call occurs [#_add_long] ; |10|
        MOVL      *–SP[6],ACC           ; |10|
        SUBB      SP,#6
        LRETR
        ; return occurs
```

6.8.4 The FUNC_EXT_CALLED Pragma

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

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

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

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma FUNC_EXT_CALLED

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

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

6.8.5 The INTERRUPT Pragma

The INTERRUPT pragma allows you to handle interrupts directly with C/C++ code. In C, the argument *func* is the name of a function. In C++, the pragma applies to the next function declared. See section 6.7.4, *The interrupt Keyword*, on page 6-13, for more details on interrupt functions in C/C++.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma INTERRUPT

6.9 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.

6.10 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 6.9, *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. The above description also applies to far variables in C, except that these variables are placed in the .econst section, for example:

```
   far const int zero=0;
```

6.11 Generating Linknames

The compiler transforms the names of externally visible identifiers when creating their linknames. The algorithm used depends on the scope within which the identifier is declared. For objects and functions, an underscore (_) is prepended to the identifier name. C++ functions have the same initial character (_) prepended, but also have the function name further mangled. Mangling is the process of embedding a function's signature (the number and types of its parameters) into its name. The mangling algorithm used closely follows that described in *The C++ Annotated Reference Manual* (ARM). Mangling allows function overloading, operator overloading, and type-safe linking.

The general form of a C++ linkname for a function named *func* is:

_ _ _**func_ _F***parmcodes*

where *parmcodes* is a sequence of letters that encodes the parameter types of func.

For this simple C++ source file:

```
int foo(int i);  //global C++ function
{
return i;
}
```

this is the resulting assembly symbol definition for foo:

```
___foo__Fi;
```

The linkname of foo is _ _ _foo_ _Fi, indicating that foo is a function that takes a single argument of type int. To aid inspection and debugging, a name demangling utility is provided (see Chapter 10, *C++ Name Demangler Utility*) that demangles names into those found in the original C++ source.

6.12 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 TMS320C28x 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 */
  ```

6.13 Changing the Language Mode (–pk, –pr, and –ps Options)

The –pk, –pr, and –ps options let you specify how the C/C++ compiler interprets your source code. You can compile your source code in the following modes:

- Normal ANSI mode
- K&R C mode
- Relaxed ANSI mode
- Strict ANSI mode

The default is normal ANSI mode. Under normal ANSI mode, most ANSI violations are emitted as errors. Strict ANSI violations (those idioms and allowances commonly accepted by C compilers, although violations with a strict interpretation of ANSI), however, are emitted as warnings. Language extensions, even those that conflict with ANSI C, are enabled.

For C++ code, ANSI mode designates the ISO/IEC 14882–1998 standard. K&R C mode does not apply to C++ code.

6.14 Enabling Strict ANSI Mode and Relaxed ANSI Mode (–ps and –pr Options)

Use the –ps option when you want to compile under strict ANSI mode. In this mode, error messages are provided when non-ANSI features are used, and language extensions that could invalidate a strictly conforming program are disabled. Examples of such extensions are the inline and asm keywords.

Use the –pr option when you want the compiler to ignore strict ANSI violations rather than emit a warning (as occurs in normal ANSI mode) or an error message (as occurs in strict ANSI mode). In relaxed ANSI mode, the compiler accepts extensions to the ANSI C standard, even when they conflict with ANSI C.

6.15 Compiler Limits

Due to the variety of host systems supported by the TMS320C28x C/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 6–3 specifies the limits that are absolute. All the absolute limits equal or exceed those required by the ANSI C standard.

Table 6–3. Absolute Compiler Limits

Description	Limit
Filename length	512 characters
Source line length; limit is after splicing of continuation lines; applies to any single macro definition or invocation	16K characters
Length of strings built from # or ##; limit is before concatenation; all other character strings are unrestricted	512 characters
Inline assembly string length	132 characters
Macro parameters	32 parameters
Macro nesting level; limit includes argument substitutions	64 levels
#include file nesting	64 levels
Conditional inclusion (#if) nesting	64 levels
Nesting of struct, union, or prototype declarations	20 levels
Function parameters	48 parameters
Array, function, or pointer derivations on a type	12 derivations
Aggregate initialization nesting	32 levels
Local initializers; limit is approximate	150 levels (approximately)
Nesting of if statements, switch statements, and loops	32 levels

Chapter 7

Run-Time Environment

This chapter describes the TMS320C28x™ C/C++ run-time environment. To ensure successful execution of C/C++ programs, it is critical that all run-time 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/C++ code.

Topic **Page**

7.1 Memory Model

The TMS320C28x 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/C++ program are placed into contiguous blocks in the appropriate memory space.

Note: The Linker Defines the Memory Map

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

The compiler produces relocatable code that allows the linker to allocate code and data into the appropriate memory spaces. For example, you can use the linker to allocate global variables into fast internal RAM or to allocate executable code into internal ROM. Each block of code or data could be allocated individually into memory, but this is not a general practice (an exception to this is memory-mapped I/O, although physical memory locations can be accessed with C pointer types).

7.1.1 Sections

The compiler produces several 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 *TMS320C28x 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 the following initialized sections: .text, .cinit, .const, .econst, .pinit, and .switch.
  - The **.text section** is an initialized section that contains all the executable code as well as constants.
  - The **.cinit section** and the **.pinit section** contain 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 **.econst section** is an initialized section that contains string constants, and the declaration and initialization of global and static variables (qualified by *far const* or the use of the large memory model) that are explicitly initialized and placed in far memory.
  - 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, .ebss, .stack, .sysmem, and .esysmem.
  - 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 **.ebss section** is an uninitialized section that reserves space for global and static variables declared as far (C only) or when the large memory model is used. 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 .ebss 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 **.esysmem section** is a uninitialized section that reserves space for dynamic memory allocation. The reserved space is used by far malloc functions. If no far 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/.ebss, .stack, and .sysmem/.esysmem 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
.pinit	ROM or RAM	0
.switch	ROM or RAM	0, 1
.const	ROM or RAM	1
.econst	ROM or RAM	1
.bss	RAM	1
.ebss	RAM	1
.stack	RAM	1
.sysmem	RAM	1
.esysmem	RAM	1

For more information about allocating sections into memory, see the introductory COFF information in the *TMS320C28x Assembly Language Tools User’s Guide*.

7.1.2 C/C++ System Stack

The C/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 run-time 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.

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.

For frames that exceed 63 words in size (the maximum reach of the SP offset addressing mode), the compiler uses XAR2 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.

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 run-time environment, causing your program to fail. Be sure to allow enough space for the stack to grow.

7.1.3 Allocating .const/.econst to Program Memory

If your system configuration does not support allocating an initialized section such as .const/.econst to data memory, then you have to allocate the .const/.econst section to load in program memory and run in data memory. At boot time, copy the .const/.econst 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:

   `ar2000 –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:

   `asm2000 –v28 boot.asm`

4) Archive the boot routine into the object library:

   `ar2000 –r rts2800.lib boot.obj`

For a .const section, 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;
               }
   ...
}
```

Similarly, for an .econst section, link with a linker command file that contains the following entries:

```
SECTIONS
{
   ...
   .econst  : load = PROG PAGE 1, run = DATA PAGE 1
              {
                 /* GET RUN ADDRESS   */
                  __econst_run = .;
                 /* MARK LOAD ADDRESS */
                  *(.ec_mark)
                 /* ALLOCATE .econst   */
                  *(.econst)
                 /* COMPUTE LENGTH    */
                  __econst_length = – .__econst_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.

7.1.4 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.

A far memory pool or far heap, is also available through several far run-time-support library functions (such as far_malloc, far_calloc, and far_realloc). The far heap is created by the linker. The linker also creates a global symbol, _ _FAR_SYSMEM_SIZE, and assigns it a value equal to the size of the far heap in words. The default size is 1k words. You can change the size of the far memory pool, at link time, with the –farheap option. Specify the size of the memory pool as a constant after the –farheap option on the linker command line.

Note: Heap Size Restriction

The near heap implementation restricts the size of the heap to 32k words. This constraint does not apply to the far heap.

7.1.5 Initialization of Variables

The C/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/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 7.8, *System Initialization*, on page 7-33.

7.1.6 Allocating Memory for Static and Global Variables

A unique, contiguous space is allocated for all external or static variables declared in a C/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/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.

7.1.7 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.

7.1.8 Character String Constants

In C/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 7.8, *System Initialization*, on page 7-33.

- 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 .string 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   .string  ”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! */
```

7.1.9 far Character String Constants

In C, a character string constant can be placed in the .econst section. When initializing a character pointer or using a character in an expression, use the far keyword. For example:

```
   far char *ptr = “abc”;
   far_strcpy (s, (far char *) “abc”);
```

far string constants are placed in the .econst section in the same manner as described in section 7.1.8, Character String Constants, on page 7-9. The far string labels have the form FSLn. This method is also used with the large memory model.

7.2 Register Conventions

Strict conventions associate specific registers with specific operations in the C/C++ environment. If you plan to interface an assembly language routine to a C/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.

7.2.1 TMS320C28x Register Use and Preservation

Table 7–1 summarizes how the compiler uses the TMS320C28x registers and shows which registers are defined to be preserved across function calls.

Table 7–1. 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
XAR0	Pointers and expressions	No	Yes
XAR1	Pointers and expressions	Yes	No
XAR2	Pointers, expressions, and frame pointer (when needed)	Yes	No
XAR3	Pointers and expressions	Yes	No
XAR4	Pointers, expressions, argument passing, and returns 16- and 22-bit pointer values from functions	No	Yes
XAR5	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

Table 7–1. Register Use and Preservation Conventions (Continued)

Register(s)	Usage	Save on Entry	Save on Call
SP	Stack pointer	†	†
T	Multiply and shift expressions	No	Yes
TL	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.

7.2.2 Status Registers

Table 7–2 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 7–2. 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.

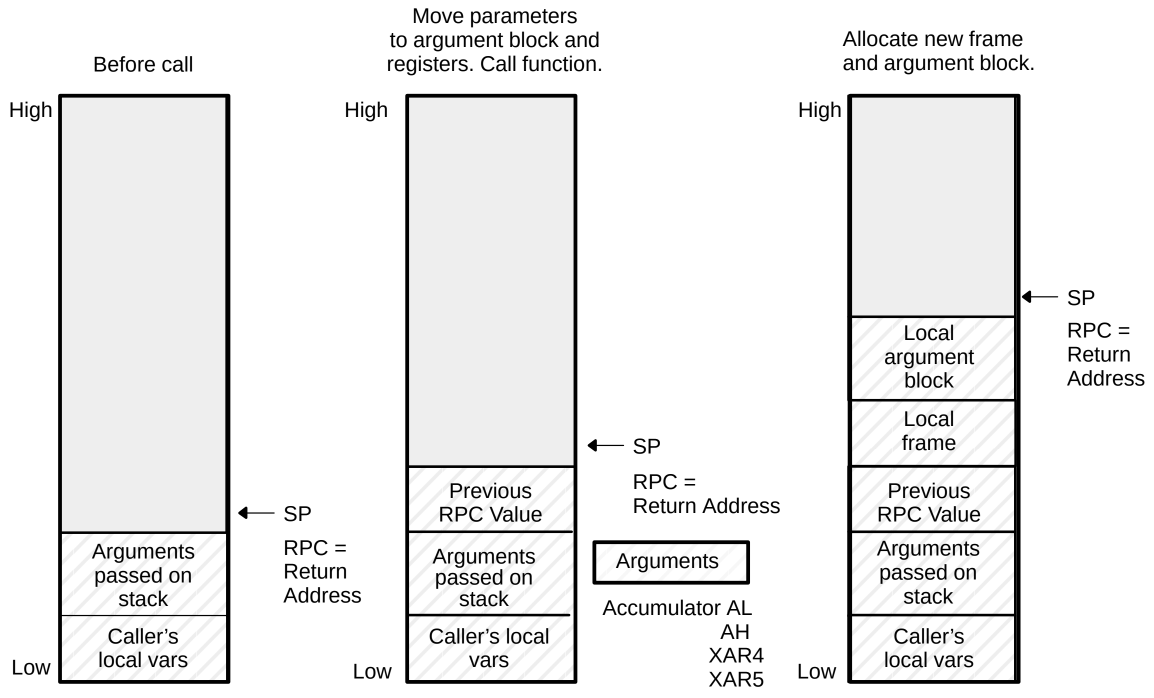
7.3 Function Calling Conventions

The C/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/C++ function must follow these rules. Failure to adhere to these rules can disrupt the C/C++ environment and cause a program to fail.

Figure 7–1 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 7–1. Use of the Stack During a Function Call



7.3.1 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 or floats) 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 XAR4 and XAR5. All other pointers are placed on the stack.

   c) Remaining 16-bit arguments are placed in the order AL, AH, XAR4, XAR5 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 are 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) ;
       XAR4   XAR5    AH/AL
func1 (long a, int b, long c) ;
       AH/AL   XAR4   stack
vararg (int a, int b, int c, ...)
        AL     AH     stack
```

5) The caller calls the function using the LCR instruction. The RPC register value is pushed on the stack. The return address is then stored in the RPC register.

7.3.2 How a Called Function Responds

A called function performs the following tasks:

1) If the called function modifies XAR1, XAR2, or XAR3, 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- or 22-bit pointer:	XAR4

   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 XAR4. 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 using the LRETR instruction. The PC is set to the value in the RPC register. The previous RPC value is popped from the stack and stored in the RPC register.

7.3.3 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 (XAR2) 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.

7.3.4 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 XAR2). 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 (XAR2). 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 XAR2 and possibly an index register, extra code and resources are required to make local accesses.

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

Some TMS320C28x 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 *TMS320C28xAssembly Language Tools User’s Guide*.

7.4 Interfacing C/C++ With Assembly Language

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

- Use separate modules of assembled code and link them with compiled C/C++ modules (see section 7.4.1, *Assembly Language Modules* on page 7-18). This is the most versatile method.
- Use inline assembly language embedded directly in the C/C++ source (see section 7.4.4, *Inline Assembly Language*, on page 7-23).
- Use intrinsics in C/C++ source to call an assembly language statement directly (see section 7.4.5, *Using Intrinsics to Access Assembly Language Statements*, on page 7-24).

7.4.1 Using Assembly Language Modules With C/C++ Code

Interfacing with assembly language functions is straightforward if you follow the calling conventions defined in section 7.3, *Function Calling Conventions*, on page 7-13, and the register conventions defined in section 7.2, *Register Conventions*, on page 7-11. C/C++ code can access variables and call functions defined in assembly language, and assembly code can access C/C++ variables and call C/C++ functions.

Follow these guidelines to interface assembly language and C/C++:

- All functions, whether they are written in C/C++ or assembly language, must follow the conventions outlined in section 7.2, *Register Conventions*, on page 7-11.
- Dedicated registers modified by a function must be preserved. Dedicated registers include:

  XAR1
  XAR2
  XAR3
  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 7.5, *Interrupt Handling*, on page 7-29.)

- ❑ When you call a C/C++ function from assembly language, load the designated registers with arguments and push the remaining arguments onto the stack as described in section 7.3.1, *How a Function Makes a Call*, on page 7-14.

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

- ❑ Since the C/C++ calling convention uses the RPC to store return values through the use of the LCR and LRETR instructions, the assembly function must follow the same convention.
- ❑ 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 7.3.2, *How a Called Function Responds*, on page 7-15.
- ❑ No assembly language module should use the .cinit section for any purpose other than autoinitialization of global variables. The C/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/C++. For example, a C/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/C++ identifier.
- ❑ Any object or function declared in assembly language that is to be accessed or called from C/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/C++ function or object from assembly language, declare the C/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 7–1 illustrates a C/C++ function called main, which calls an assembly language function called asmfunc. The asmfunc function takes its single argument, adds it to the C/C++ global variable called gvar, and returns the result.

Example 7–1. An Assembly Language Function

(a) C++ program

```
extern “C”{
extern int asmfunc(int a); /* declare external asm function */
int gvar = 0;              /* define global variable        */
}


void main()
{
   int i = 5;

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

(b) Assembly language program

```
        .global _gvar
        .global _asmfunc

_asmfunc:
        MOVZ    DP,#_gvar
        ADDB    AL,#5
        MOV     @_gvar,AL
        LRETR
```

In the C++ program in Example 7–1, the extern “C” declaration tells the compiler to use C naming conventions (i.e., no name mangling). When the linker resolves the .global _asmfunc reference, the corresponding definition in the assembly file will match.

The parameter i is passed in register AL.

7.4.2 Accessing Assembly Language Global Variables

It is sometimes useful for a C/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 7–2 shows an example that accesses a variable defined in .bss.

Example 7–2. Accessing a Variable Defined in .bss From C/C++

(a) C/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 7–3 shows an example that accesses a variable that is not defined in .bss.

Example 7–3. 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
```

7.4.3 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/C++ only with the use of special operators.

For normal variables defined in C/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/C++ is &x.

You can use casts and #defines to ease the use of these symbols in your program, as in Example 7–4.

Example 7–4. 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 7–4, int is used. You can reference linker-defined symbols in a similar manner.

7.4.4 Using Inline Assembly Language

Within a C/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/C++. When you use the asm statement, be extremely careful not to disrupt the C/C++ environment. The compiler does not check or analyze the inserted instructions.

Inserting jumps or labels into C/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/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.

If an assembly statement is added that accesses the IER or IFR control registers, it is preferable to use the cregister keyword in C/C++ for these accesses. For more information, see section 6.7.3, *The cregister Keyword*, on page 6-12.

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”);
```

7.4.5 Using Intrinsics to Access Assembly Language Statements

The TMS320C28x compiler recognizes a number of intrinsic operators. Intrinsics are used like functions and produce assembly language statements that would otherwise be inexpressible in C/C++. You can use C/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 intrinsics listed in Table 7–3 are included. They correspond to the indicated TMS320C28x assembly language instruction(s). See the *TMS320C28x CPU and Instruction Set Reference Guide* for more information.

Table 7–3. TMS320C28x C/C++ Compiler Intrinsics

C/C++ Compiler Intrinsic	Assembly Instruction(s)	Description
int **__abs16_sat(**int *src* **)**	**SETC OVM** **MOV AH,** *src* **ABS ACC** **MOV** *dst*, **AH** **CLRC OVM**	Clear the OVM status bit. Load src into AH. Take absolute value of ACC. Store AH into dst. Clear the OVM status bit.
long **__addcu(**long *src1***,** unsigned int *src2***);**	**ADDCU ACC,** {*mem* | *reg*}	The contents of *src2* and the value of the carry bit are added to ACC. The result is in ACC.
long **__mpy(**int *src1***,** int *src2***);**	**MPY ACC,** *src1*, *#src2*	Move *src1* to the T register. Multiply T by a 16-bit immediate (*src2*). The result is in ACC.
int **__mov_byte(**int **src*, unsigned int *n***);**	**MOVB AX.LSB,*+XARx[***n***]** or **MOVZ AR0/AR1,** @*n* **MOVB AX.LSB,*XARx[AR0/AR1]**	Return the 8-bit nth element of a byte table pointed to by *src*.

Table 7–3. TMS320C28x C/C++ Compiler Intrinsics (Continued)

C/C++ Compiler Intrinsic	Assembly Instruction(s)	Description
long __**mpyb(**int *src1***,** uint *src2***);**	**MPYB** {**ACC** | **P**}**, T,** *#src2*	Multiply *src1* (the T register) by an unsigned 8-bit immediate (*src2*). The result is in ACC or P.
long __**mpy_mov_t(**int *src1***,** int *src2*, int **dst2***);**	**MPY ACC, T,** *src2*	Multiply *src1* (the T register) by *src2*. The result is in ACC. Move *src1* to *dst2*.
	MOV *dst2***, T**	
unsigned long __**mpyu(**uint *src1***,** uint *src2***);**	**MPYU** {**ACC** | **P**}**, T**, *src2*	Multiply *src1* (the T register) by *src2*. Both operands are treated as unsigned 16-bit numbers. The result is in ACC or P.
long __**mpyxu(**int *src1*, uint *src2***);**	**MPYXU ACC, T,** *{mem|reg}*	The T register is loaded with src1. Src2 is referenced by memory or loaded into a register. The result is in ACC.
long __**qmpy32(**long *src32a***,** long *src32b***,** int *q*);	**CLRC OVM**	Extended precision DSP Q math. Different code is generated based on the value of q.
	SPM – 1	
	MOV T, *src32a* **+ 1**	
	MPYXU P, T, *src32b* **+ 0**	
	MOVP T, *src32b* **+ 1**	
	MPYXU P, T, *src32a* **+ 0**	
	MPYA P, T, *src32a* **+ 1**	
	; if q = 31,30	
	SPM q – 30	
	SFR ACC, #45 – q	
	ADDL ACC, P	
	; if q = 29	
	SFR ACC, #16	
	ADDL ACC, P	
	;if q = 28 through 24	
	SPM q – 30	
	SFR ACC, #16	
	SFR ACC, #29 – q	
	ADDL ACC, P	
	;if q = 23 through 13	
	SFR ACC, #16	
	ADDL ACC, P	
	SFR ACC, #29 – q	
	;if q = 12 through 0	
	SFR ACC, #16	
	ADDL ACC, P	
	SFR ACC, #16	
	SFR ACC, #13 – q	

Table 7–3. TMS320C28x C/C++ Compiler Intrinsics (Continued)

C/C++ Compiler Intrinsic	Assembly Instruction(s)	Description
long __**qmpy32by16(**long *src32*, int src16**,** int *q***);**	**CLRC OVM** **MOV T,** *src16* **+ 0** **MPYXU P, T,** *src32* **+ 0** **MPY P, T,** *src32* **+ 1** **; if q = 31,30** **SPM q – 30** **SFR ACC, #46 – q** **ADDL ACC, P** **; if q = 29 through 14** **SPM 0** **SFR ACC, #16** **ADDL ACC, P** **SFR ACC, #30 – q** **;if q = 13 through 0** **SPM 0** **SFR ACC, #16** **ADDL ACC, P** **SFR ACC, #16** **SFR ACC, #14 – q**	Extended precision DSP Q math. Different code is generated based on the value of q.
long __**rol(**long *src***);**	**ROL ACC**	Rotate ACC left.
long __**ror(**long *src***);**	**ROR ACC**	Rotate ACC right.
void *result = __**rpt_mov_imm;**	**MOV** *result***,** *dst* **MOV AR***x***,** *dst* **RPT** *#count* **|| MOV *XAR***x***++,** *#src*	Move the *dst* register to the *result* register. Move the *dst* register to a temp (ARx) register. Copy the immediate *src* to the temp register *count* + 1 times. ❑ *src* must be a 16-bit immediate. ❑ *count* can be an immediate from 0 to 255 or a variable.
far void *result = __**rpt_mov_imm_far** **(**far void ***dst*, int *src*, int *count***);**	**MOVL** *result***,** *dst* **MOVL XAR***x***,** *dst* **RPT** *#count* **|| MOV *XARx++,** *#src*	Move the dst register to the result register. Move the dst register to a temp (XARx) register. Copy the immediate src to the temp register count + 1 times. ❑ *src* must be a 16-bit immediate. ❑ *count* can be an immediate from 0 to 255 or a variable.

Table 7–3. TMS320C28x C/C++ Compiler Intrinsics (Continued)

C/C++ Compiler Intrinsic	Assembly Instruction(s)	Description
int **__rpt_norm_inc(**long *src***,** int *dst***,** int *count***);**	**MOV AR***x***,** *dst* **RPT** *#count* **|| NORM ACC, AR***x***++**	Repeat the normalize accumulator value *count* + 1 times. *count* can be an immediate from 0 to 255 or a variable.
int **__rpt_norm_dec(**long *src***,** int *dst***,** int *count***);**	**MOV AR***x***,** *dst* **RPT** *#count* **|| NORM ACC, AR***x***++**	Repeat the normalize accumulator value *count* + 1 times. *count* can be an immediate from 0 to 255 or a variable.
int **__rpt_rol(**long *src***,** int *count***);**	**RPT** *#count* **|| ROL ACC**	Repeat the rotate accumulator left *count* + 1 times. The result is in ACC. *count* can be an immediate from 0 to 255 or a variable.
int **__rpt_ror(**long *src***,** int *count***);**	**RPT** *#count* **|| ROR ACC**	Repeat the rotate accumulator right *count* + 1 times. The result is in ACC. *count* can be an immediate from 0 to 255 or a variable.
long **__rpt_subcu(**long *dst***,** int *src1*, int *rpt_cnt***);**	**SUBCU, ACC,** [*mem*|*reg*]	Use *rpt_cn*t 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 **__rpt_subcu(**long *dst***,** int *src***,** int *count***);**	**RPT** *count* **|| SUBCU ACC,** *src*	The *src* operand is referenced from memory or loaded into a register and used as an operand to the SUBCU instruction. The result is in ACC. *count* can be an immediate from 0 to 255 or a variable. The instruction repeats *count* + 1 times.
long **__sat(**long *src***);**	**SAT ACC**	Load ACC with 32-bit src. The result is in ACC.
long **__sat32(**long *src,* long *limit***);**	**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. Limit value is either referenced from memory or loaded into the P register. The result is in ACC.

Table 7–3. TMS320C28x C/C++ Compiler Intrinsics (Continued)

C/C++ Compiler Intrinsic	Assembly Instruction(s)	Description
long _ _**sathigh16(**long *src*, int *limit* **);**	**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. Limit value is either referenced from memory or loaded into register. The result is in ACC. The result can be right shifted and stored into an int. For example: `ivar = __sathigh16(lvar, mask) >> 6;`
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.
long __**sbbu(**long *src1***,** uint *src2***);**	**SBBU ACC,** *src2*	Subtract *src2* + logical inverse of C from ACC (*src1*). The result is in ACC.
long __**subcu(**long *src1***,** int *src2* **);**	**SUBCU ACC,** *src2*	Subtract *src2* shifted left 15 from ACC (*src1*). The result is in ACC.
if **(**_ _**tbit**(int *src***,** int *bit* **))**	**TBIT** *src***,** *#bit*	SET TC status bit if specified bit of *src* is 1.

7.5 Interrupt Handling

As long as you follow the guidelines in this section, C/C++ code can be interrupted and returned to without disrupting the C/C++ environment. When the C/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/C++ environment and can be easily implemented with asm statements.

7.5.1 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/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 run-time stack is set up; therefore, you *cannot allocate local variables*, and you *cannot save any information on the run-time 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.

7.5.2 Using C/C++ Interrupt Routines

If a C/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/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/C++ interrupt routine is like any other C/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/C++ functions by using the interrupt pragma or the interrupt keyword. For information about the interrupt pragma, see section 6.8.5, *The INTERRUPT Pragma*, on page 6-28. For information about the interrupt keyword, see section 6.7.4, *The interrupt Keyword*, on page 6-13.

7.6 Integer Expression Analysis

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

7.6.1 Integer Operations Evaluated With RTS Calls

The TMS320C28x does not directly support some C/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 (rts2800.lib and rts.src) in the toolset.

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

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

7.6.2 C/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/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
```

7.7 Floating-Point Expression Analysis

The TMS320C28x C/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 run-time-support library, rts2800.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.

7.8 System Initialization

Before you can run a C/C++ program, the C/C++ run-time environment must be created. This task is performed by the C/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 (rts2800.lib) and must be linked with the C/C++ object modules. This occurs by default when you use the –c or –cr option in the linker and include rts2800.lib as a linker input file. When C/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/C++ environment:

1) Reserves space for the run-time stack and sets up the initial stack pointer value.
2) Initializes status bits and registers to expected values.
3) 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 autoinitialization model, a loader performs this step before the program runs (it is not performed by the boot routine).
4) Executes the global constructors found in the .pinit section. For more information, see section 7.8.3, *Global Constructors*.
5) Calls the function main to run the C/C++ program.

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

7.8.1 Run-Time Stack

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

The code does not check to see if the run-time stack overflows. Stack overflow occurs when the stack grows beyond the limits of the memory space that was allocated for it. Be sure to allocate adequate memory for the stack.

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

7.8.2 Automatic Initialization of Variables

Some global variables must have initial values assigned to them before a C/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 ANSI C, global and static variables that are not explicitly initialized are set to 0 before program execution. The C28x C/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 7.8.5, *Autoinitialization of Variables at Run Time*, on page 7-38 and 7.8.6, *Autoinitialization of Variables at Load Time*, on page 7-39).

7.8.3 Global Constructors

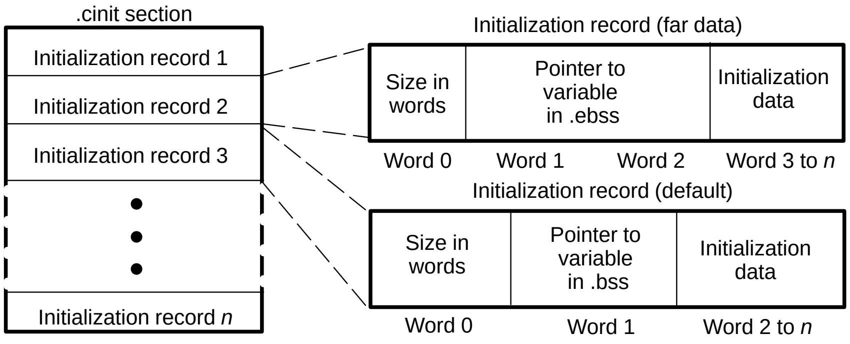
All global C++ variables that have constructors must have their constructor called before main( ). The compiler builds a table of global constructor addresses that must be called, in order, before main( ) in a section called .pinit. The linker combines the .pinit section from each input file to form a single table in the .pinit section. The boot routine uses this table to execute the constructors.

7.8.4 Initialization Tables

The generation of .cinit records efficiently creates records that take into account far data addresses.

The tables in the .cinit section consist of variable-size initialization records. Figure 7–2 shows the format of the .cinit section and the initialization records.

Figure 7–2. Format of Initialization Records in the .cinit Section (Default and far Data)



The fields of an initialization record contain the following information:

- The first field is the size in words of the initialization data for the variable. A negative value for this field denotes that the variable's address is far.
- The second field is the starting address of the variable in the .bss section into which the initialization data must be copied. If the variable is far, the field points to the variable's space in .ebss. For far data the second field requires two words to hold the address.
- The third field contains the data that is copied into the variable to initialize it.

Note: Initializing Variables

The compiler only supports initializing objects that are 32k words in size or less.

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;
far    int j[2] = { 1,2};
```

In this case, the initialization table entries are as follows:

```
      .global   _i
      .bss      _i,1,1,0
      .global   _j
_j:   .usect    .ebss,2,1,0

      .sect     ”.cinit”
      .align    1
      .field          1,16
      .field          _i+0,16
      .field          23,16           ; _i @ 0


      .sect     ”.cinit”
      .align    1
      .field          –IR_1,16
      .field          _j+0,32
      .field          1,16            ; _j[0] @ 0
      .field          2,16            ; _j[1] @ 16
IR_1: .set      2
```

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.

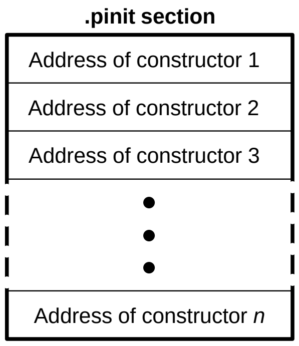
When you use the –c or –cr linker option, the linker links together the .cinit sections from all the C modules and appends a null word to the end of the composite .cinit section. This terminating record appears as a record with a size field of 0, marking the end of the initialization tables.

Note: Initialization of const Variables

Variables qualified as const are initialized differently; see section 6.9, *Initializing Static and Global Variables*, on page 6-30.

The table in the .pinit section simply consists of a list of addresses of constructors to be called (see Figure 7–3). The constructors appear in the table in the order they must be executed.

Figure 7–3. Format of the .pinit Section



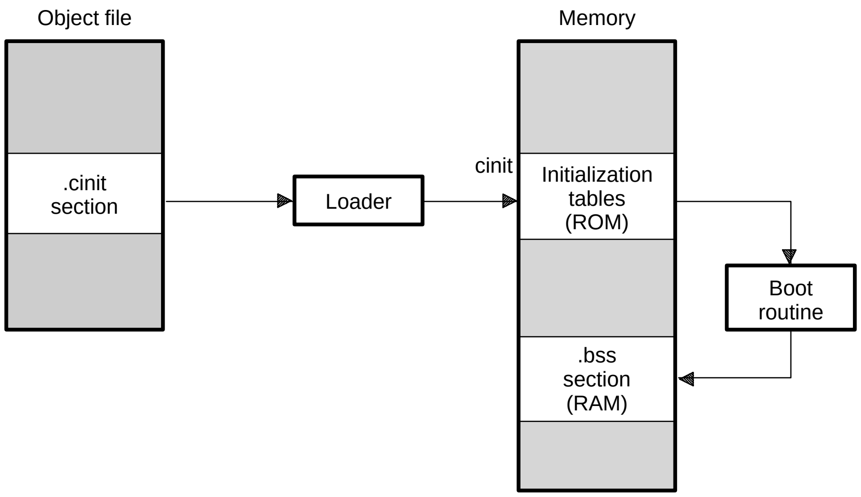
7.8.5 Autoinitialization of Variables at Run Time

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/C++ boot routine copies data from the tables (pointed to by .cinit) into the specified variables in the .bss section. This allows initialization data to be stored in ROM and copied to RAM each time the program starts.

Figure 7–4 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into ROM.

Figure 7–4. Autoinitialization at Run Time



7.8.6 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.

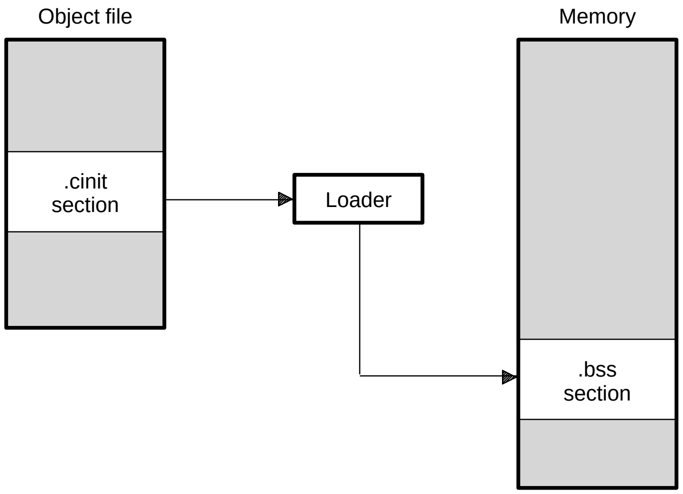
When you use the –cr linker option, the linker sets the STYP_COPY bit in the .cinit section's header. This tells the loader not to load the .cinit section into memory. (The .cinit section occupies no space in the memory map.) The linker also sets the cinit symbol to –1 (normally, cinit points to the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

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

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

Figure 7–5 illustrates the autoinitialization of variables at load time.

Figure 7–5. Autoinitialization at Load Time



Chapter 8

Run-Time-Support Functions

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

In addition to the ANSI-specified functions, the run-time-support library includes routines that give you processor-specific commands and direct C language I/O requests. The library is also extended to include far dynamic memory management routines (for C/C++) and far versions of the ANSI routines (for C only).

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 9, *Library-Build Utility*); include that library when you link your C/C++ program.

Topic **Page**

8.1 Libraries

The following libraries are included with the TMS320C28x C/C++ compiler:

- *rts2800.lib* contains the ANSIC/C++ run-time-support object library
- *rts.src* contains the source for the ANSIC/C++ run-time-support routines.
- *rts2800_ml.lib* contains the C/C++ large memory model run-time-support object library

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

8.1.1 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 *TMS320C28x Assembly Language Tools User’s Guide*.

8.1.2 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:

```
ar2000 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:

```
cl2000 –v28 –options atoi.c strcpy.c        ; recompile
ar2000 r rts2800.lib atoi.obj strcpy.obj    ; rebuild library
```

You can also build a new library this way, rather than rebuilding into rts2800.lib. For more information about the archiver, see the *TMS320C28x Assembly Language Tools User’s Guide*.

8.1.3 Building a Library With Different Options

You can create a new library from rts.src by using the library-build utility mk2000 –v28. For example, use this command to build an optimized run-time-support library:

```
mk2000 –v28 ––u –o2 rts.src –l rts2800.lib
```

The ––u option tells the mk2000 –v28 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 TMS320C28x. The use of the (–o2) option does not affect compatibility with code compiled without these options. For more information on the library-build utility, see Chapter 9.

8.2 Far Memory Support

As described in section 6.7.6, *The Far Keyword*, on page 6-16, the C/C++ compiler extends the C language (not C++) by providing the far keyword. The far keyword is used to access 22 bits of address space.

The run-time-support library is extended to include far versions of the ANSI routines and far dynamic memory management routines.

The large memory model C++ run-time-support library automatically supports far memory since all pointers are 22 bits.

8.2.1 Far Versions of Run-Time-Support Functions

To provide far support to the C run-time library, a far version is defined for most of the run-time-support functions that either have a pointer argument or returns a pointer. In the following example, the atoi() run-time-support function takes a string (pointer to char) argument and returns the integer value represented by the string.

```
#include <stdlib.h>
char * st = ″1234″;
.
.
.
.
int    ival = atoi(st);   /* ival is 1234 */
```

The far version of the atoi() function, far_atoi(), is defined to take a far string (pointer to far char) argument and return the integer value.

```
#include <stdlib.h>
far char * fst = ″1234″;
.
.
.
.
int    ival = far_atoi(fst); /* ival is 1234 */
```

The far version of the run-time-support function performs the same operations except that it accepts or returns pointer to far objects. Table 8–3(j) , *Summary of far versions*, on page 8-35, lists all the far functions in the run-time-support library.

8.2.2 Global and Static Variables in Run-Time-Support

The run-time-support library uses several global and static variables. Some of them are for internal use and others are for passing status and other information to the user, as in the case of global variable errno defined in stderr.h. By default, these variables are placed in the .bss section and considered near objects. For more information, see section 7.1.1, *Section*, on page 7-3.

The C I/O functions do not have corresponding far versions. Also, the functions that use the C I/O functions do not have corresponding far versions.

You can place global and static variables in far memory (.ebss section) by defining _FAR_RTS (–d_FAR_RTS) when building the C run-time library (not C++). The following entry on the command line builds the C run-time-support library with global and static variables in far memory. The header file, linkage.h, defines the necessary macros for the _FAR_run-time-support build.

```
mk2000 –v28 –d_FAR_RTS rts.src –l rts2800.lib
```

8.2.3 Far Dynamic Memory Allocation in C

You can allocate far memory dynamically at run time. The far memory is allocated from a global pool or far heap that is defined in the .esysmem section. For more information, see section 7.1.4, *Dynamic Memory Allocation*, on page 7-7.

The run-time-support library includes the following functions that allow you to dynamically allocate far memory at run time:

far void * far_malloc (unsigned long *size*);
far void * far_calloc (unsigned long *num*, unsigned long *size*);
far void * far_realloc (far void **ptr*, unsigned long *size*);
void far_free (far void **ptr*);
long far_free_memory(void);
long far_max_free(void);

The following C code allocates memory for 100 far objects and deallocates the memory at the end.

```
#include <stdlib.h>

struct big {
   int a, b;
   char c[80];
};

int main()
{
   far struct big *table;
   table=(far struct big *)far_malloc(100*sizeof(struct
big));
   .
   .
   /* use the memory here */
   .
   .
   far_free(table);
   /* exit code here */
}
```

8.2.4 Building a Large Memory Model Run-Time-Support Library

You can build a large memory model C or C++ library by using the –ml shell option. The run-time-support library source code supports the large memory model through conditional compilation. When compiling the run-time-support libraries, the LARGE_MODEL symbol must be defined. For example, to build a large memory model C++ library use this command:

```
mk2000 –v28 –ml –DLARGE_MODEL –o rtscpp.src –l rtscpp_ml.lib
```

8.2.5 Far Dynamic Memory Allocation in C++

In C++ mode the compiler does not support the far keyword. Far intrinsics are provided to access far memory if the large memory model is not used. For more information, see section 6.7.8, *Using Intrinsics to Access Far Memory in C++*, on page 6-19. You can dynamically define objects in far memory and access them using the far intrinsics. The data type long is used to hold the far pointer.

The C++ run-time-support library provides the same set of dynamic far memory allocation functions as C run-time-support library. The C++ functions use the data type long to accept as return the far pointers, so that the memory can be accessed using far intrinsics. The C++ dynamic far memory allocation functions are listed below:

long std::far_malloc (unsigned long *size*);
long std::far_calloc (unsigned long *num*, unsigned long *size*);
long std::far_realloc (long *ptr*, unsigned long *size*);
void std::far_free (long *ptr*);
long std::far_free_memory (void);
long std::far_max_free (void);

The following C++ code allocates memory for 100 far objects and deallocates the memory at the end.

```
#include <cstdlib>

struct big {
   int a, b;
   char c[80];
};

int main()
{
   long table;//Note–use of long to hold address.
   table = std::far_malloc(100 * sizeof(struct big));
   .
   .
   /* use the memory here using intrinsic*/
   .
   .
   std::far_free(table);
   /* exit code here */
}
```

Note: Using Far Intrinsics

The file, farmemory.cpp, in rts.src implements far dynamic memory allocation functions using far intrinsics. You can refer to this file as an example about how to use far intrinsics.

8.3 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 function.
- Allow for 320 bytes of heap space for each I/O stream used in your program. A stream is a source or destination of data that is associated with a peripheral, such as a terminal or keyboard. Streams are buffered using dynamically allocated memory that is taken from the heap. More heap space may be required to support programs that use additional amounts of dynamically allocated memory (calls to malloc()). To set the heap size, use the –heap option when linking. See section 4.4, *Linker Options,* on page 4-6, for more information about the –heap option.

For example, assume the following program is 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:

```
cl2000 –v28 main.c –z –l rts2800.lib –o main.out
```

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

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 functions also offer facilities to perform I/O on a user-specified device.

8.3.1 Overview of Low-Level I/O Implementation

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

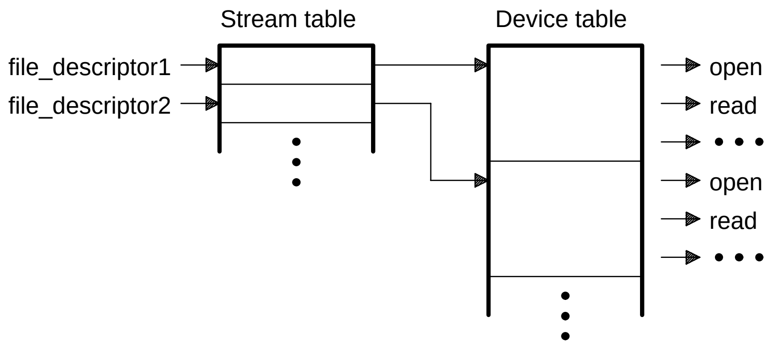
The high-level functions are the standard C library of stream I/O routines (printf, scanf, fopen, getchar, etc.). These routines map an I/O request to one or more of the I/O commands that are handled by the low-level shell.

The low-level functions are composed of basic I/O functions: OPEN, READ, WRITE, CLOSE, LSEEK, RENAME, and UNLINK. These low-level functions 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.

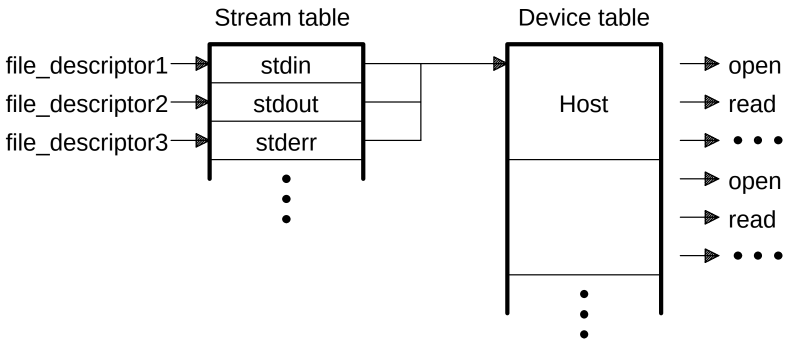
The data structures interact as shown in Figure 8–1.

Figure 8–1. 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 8–2. 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 C I/O library includes the device drivers necessary to perform C I/O on the host on which the debugger is running.

The specifications for writing device-level routines so that they interface with the low-level routines are described in section 8.3.2, *Adding a Device for C I/O*, on page 8-15. You should write each function to 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 for C

#include <stdio.h>
#include <file.h>

int close(int *file_descriptor***);**

Syntax for C++

#include <cstdio>
#include <file.h>

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 function returns one of the following values:

0	if successful
–1	if fails

lseek — *Set File Position Indicator*

Syntax for C

#include <stdio.h>
#include <file.h>

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

Syntax for C++

#include <cstdio>
#include <file.h>

long std::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 function returns one of the following values:

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

open — *Open File or Device For I/O*

Syntax for C

#include <stdio.h>
#include <file.h>

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

Syntax for C++

#include <cstdio>
#include <file.h>

int std::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  (0x200)  /* open with file create */
O_TRUNC  (0x400)  /* 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. However, 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 *file_descriptor* is the stream number assigned by the low-level routines that the device-level driver should associate with the opened file or device.

Return Value The function returns one of the following values:

>= 0	if successful
< 0	if fails

read — *Read Characters From Buffer*

Syntax for C

#include <stdio.h>
#include <file.h>

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

Syntax for C++

#include <cstdio>
#include <file.h>

int std::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 for C

#include <stdio.h>
#include <file.h>

int rename(char ****old_name*, **char ****new_name*);**

Syntax for C++

#include <cstdio>
#include <file.h>

int std::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 the rename is successful
Nonzero	if fails

unlink — *Delete File*

Syntax for C

#include <stdio.h>
#include <file.h>

int unlink(char ****path*);**

Syntax for C++

#include <cstdio>
#include <file.h>

int std::unlink(char ****path*);**

Description

The unlink function deletes the file specified by *path*.

The *path* is the filename of the file to be deleted, 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 for C

#include <stdio.h>
#include <file.h>

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

Syntax for C++

#include <cstdio>
#include <file.h>

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

8.3.2 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 8.3.1, *Overview of Low-Level I/O Implementation*, on page 8-9.

Note: Use Unique Function Names

The function names open(), close(), read(), etc. have been 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 of the add_device function, see page 8-40.

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().

8.4 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 run-time-support functions:

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

In addition to the ANSI C header files, the following C++ header files are included:

cassert	cmath	cstdlib	stdexcept
cctype	csetjmp	cstring	typeinfo
cerrno	cstdarg	ctime	
cfloat	cstddef	exception	
climits	cstdio	new	

The C++ header files beginning with a **c** correspond to the ANSI C headers without the c (i.e., cerrno corresponds to errno.h). The C++ header files beginning with a c provide the same functionality as their corresponding ANSI C header file, but introduce all types and functions into the std namespace. You can still use the ANSI C header files with your C++ applications, but their use is deprecated.

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, assuming a C application, 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 8.4.1 through 8.4.6 describe the header files that are included with the C/C++ compiler. Section 8.5, *Summary of Run-Time-Support Functions and Macros*, on page 8-27, lists the functions that these headers declare.

8.4.1 Diagnostic Messages (assert.h/cassert)

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

The assert.h/cassert header refers to another macro named NASSERT. If you have defined NASSERT as a macro name when you include assert.h/cassert, assert assumes that the runtime expression to be tested is true. This allows the optimizer to remove the assert macro as well as to use the truth of the expression to optimize the remainder of the code.

The assert macro is defined as follows:

```
#ifdef NDEBUG
#define assert(ignore) ((void)0)

#elseif defined (NASSERT)
#define assert (expression)   _nassert(expression)

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

8.4.2 Character-Typing and Conversion (ctype.h/cctype)

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

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

The ctype.h/cctype 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 passed to one of these macros 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*).

See Table 8–3 (b) on page 8-27 for a list of these character-typing and conversion functions.

8.4.3 Error Reporting (errno.h/cerrno)

The errno.h/cerrno header declares the errno variable. The errno variable declares errors in the 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)

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/cerrno and defined in errno.c.

8.4.4 Limits (float.h/cfloat and limits.h/climits)

The float.h/cfloat and limits.h/climits headers define macros that expand to useful limits and parameters of the TMS320C28x's numeric representations. Table 8–1 and Table 8–2 list these macros and their associated limits.

Table 8–1. Macros That Supply Integer Type Range Limits (limits.h)

Macro	Value	Description
CHAR_BIT	16	Number of bits in type char
SCHAR_MIN	(–SCHAR_MAX –1)	Minimum value for a signed char
SCHAR_MAX	32767	Maximum value for a signed char
UCHAR_MAX	65535	Maximum value for an unsigned char
CHAR_MIN	SCHAR_MIN	Minimum value for a char
CHAR_MAX	SCHAR_MAX	Maximum value for a char
SHRT_MIN	–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	32767	Maximum value for an int
UINT_MAX	65535	Maximum value for an unsigned int
LONG_MIN	(–LONG_MAX – 1)	Minimum value for a long int
LONG_MAX	2147483647	Maximum value for a long int
ULONG_MAX	4294967295	Maximum value for an unsigned long int

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

Table 8–2. 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 6 6	Number of decimal digits of precision for a float, double, or long double
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	24 24 24	Number of base-FLT_RADIX digits in the mantissa of a float, double, or long double
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	–125 –125 –125	Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized float, double, or long double
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	128 128 128	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 1.19209290e–07 1.19209290e–07	Minimum positive float, double, or long double number *x* such that 1.0 + x ≠ 1.0
FLT_MIN DBL_MIN LDBL_MIN	1.17549435e–38 1.17549435e–38 1.17549435e–38	Minimum positive float, double, or long double
FLT_MAX DBL_MAX LDBL_MAX	3.40282347e+38 3.40282347e+38 3.40282347e+38	Maximum float, double, or long double
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	–37 –37 –37	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 38 38	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.

8.4.5 Floating-Point Math (math.h/cmath)

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

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

These functions are listed in Table 8–3(c) on page 8-28.

8.4.6 Nonlocal Jumps (setjmp.h/csetjmp)

The setjmp.h/csetjmp header defines one type, one macro, and one function for bypassing the normal function call and return discipline. These are listed in Table 8–3(d) on page 8-29 and 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

setjmp.h also declares the far version of jmp_buf, setump and longjmp as far_jmp_buf, far_setjmp, and far_longjmp, respectively.

8.4.7 Variable Arguments (stdarg.h/cstdarg)

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

- The three macros are *va_start*, *va_arg*, and *va_end*. These macros are used when the number and type of arguments 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/cstdarg to step through its argument list at run time when the function that is using the macro 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.

These functions are listed in Table 8–3(e) on page 8-29.

8.4.8 Standard Definitions (stddef.h/cstddef)

The stddef.h/cstddef header defines two types and two 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*, which expands to a null pointer constant (0)
- *offsetof(type, identifier)*, which expands to an integer that has type *size_t*. The result is the value of an offset in bytes to a structure member (identifier) from the beginning of its structure (type).

These types and macros are used by several of the run-time-support functions.

8.4.9 Input/Output Functions (stdio.h/cstdio)

The stdio.h/cstdio header defines seven macros, two types, a structure, and a number of functions. The types and structure are:

- *size_t*, an unsigned integer type that is the data type of the *sizeof* operator. The original declaration is in stddef.h.
- *fpos_t*, an unsigned long type that can uniquely specify every position within a file
- *FILE*, a structure that records all the information necessary to control a stream

The macros are:

- *NULL*, which expands to a null pointer constant(0). The original declaration is in stddef.h. It will not be redefined if it has already been defined.
- *BUFSIZ*, which expands to the size of the buffer that setbuf() uses
- *EOF*, which is the end-of-file marker
- *FOPEN_MAX*, which expands to the largest number of files that can be open at one time
- *FILENAME_MAX*, which expands to the length of the longest file name in characters
- *L_tmpnam*, which expands to the longest filename string that tmpnam() can generate
- *TMP_MAX*, a macro that expands to the maximum number of unique filenames that tmpnam() can generate

The functions are listed in Table 8–3(f) on page 8-29.

8.4.10 General Utilities (stdlib.h/cstdlib)

The stdlib.h header declares two types, one macro, and several common library functions. 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 macro, *RAND_MAX*, is the maximum random number the rand function will return.

The common library functions are:

- *Memory management* functions that allow you to allocate and deallocate packets of memory. By default, these functions can use 2K bytes of memory. You can change this amount at link time by invoking the linker with the –heap option and specifying the desired heap size as a constant directly after the option.
- *String conversion* functions that convert strings to numeric representations
- *Searching* and *sorting* functions that allow you to search and sort arrays
- *Sequence-generation* functions that allow you to generate a pseudorandom sequence and allow you to choose a starting point for a sequence
- *Program-exit* functions that allow your program to terminate normally or abnormally
- *Integer arithmetic* that is not provided as a standard part of the C language

The functions are listed in Table 8–3(g) on page 8-32.

stdlib.h also includes the declaration for the C far versions of the required general utilities functions. The functions are listed in Table 8–3(j) on page 8-35.

8.4.11 String Functions (string.h/cstring)

The string.h/cstring header declares standard functions that allow you to perform the following tasks with character arrays (strings):

- Move or copy entire strings or portions of strings
- Concatenate strings
- Compare strings
- Search strings for characters or other strings
- Find the length of a string

In C/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/cstring allow you to perform corresponding operations on arbitrary sequences of bytes (data objects) where a 0 value does not terminate the object. These functions have names such as mem*xxx*.

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

The functions are listed in Table 8–3(h) on page 8-33.

The string.h header also includes the far version of the C string functions. The functions are listed in Table 8–3(j) on page 8-35.

8.4.12 Time Functions (time.h/ctime)

The time.h/ctime header declares one macro, several types, and functions that manipulate dates and times. Times are represented in two 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                */
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.

Local time can be adjusted for daylight savings time. Obviously, local time depends on the time zone. The time.h/ctime header declares 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 runtime 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 time.h/ctime functions are two system functions: clock and time. Time provides the current time (in time_t format), and clock provides the system time (in arbitrary units). The value returned by clock can be divided by the macro CLOCKS_PER_SEC to convert it to seconds. Because 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.

The functions are listed in Table 8–3(i) on page 8-35.

Note: Writing Your Own Clock Function

The clock function is host-system specific, so you must write your own clock function. You must also define the CLOCKS_PER_SEC macro according to the units of your clock so that the value returned by clock()—number of clock ticks—can be divided by CLOCKS_PER_SEC to produce a value in seconds.

8.4.13 Exception Handling (exception and stdexcept)

Exception handling is not supported. The exception and stdexcept include files (for C++ only) are empty.

8.4.14 Dynamic Memory Management (new)

The new header (for C++ only) defines functions for new, new[ ], delete, delete[ ], and their placement versions.

The type new_handler and the function set_new_handler( ) are also provided to support error recovery during memory allocation.

8.4.15 Runtime Type Information (typeinfo)

The typeinfo header (for C++ only) defines the type_info structure, which is used to represent C++ type information at run time.

8.5 Summary of Run-Time-Support Functions and Macros

Table 8–3 summarizes the run-time-support header files (in alphabetical order) provided with the TMS320C28x ANSI C/C++ compiler. Most of the functions described are per the ANSI C standard and behave exactly as described in the standard.

The functions and macros listed in Table 8–3 are described in detail in section 8.6, *Description of Run-Time-Support Functions and Macros*, on page 8-38. For a complete description of a function or macro, see the indicated page.

Table 8–3. Summary of Run-Time-Support Functions and Macros

(a) Error message macro (assert.h/cassert)

Macro	**Description**	**Page**
void **assert**(int expression);	Inserts diagnostic messages into programs	8-43

(b) Character-typing conversion functions (ctype.h/cctype)

Function	**Description**	**Page**
int **isalnum**(int c);	Tests c to see if it is an alphanumeric ASCII character	8-70
int **isalpha**(int c);	Tests c to see if it is an alphabetic ASCII character	8-70
int **isascii**(int c);	Tests c to see if it is an ASCII character	8-70
int **iscntrl**(int c);	Tests c to see if it is a control character	8-70
int **isdigit**(int c);	Tests c to see if it is a numeric character	8-70
int **isgraph**(int c);	Tests c to see if it is any printing character except a space	8-70
int **islower**(int c);	Tests c to see if it is a lowercase alphabetic ASCII character	8-70
int **isprint**(int c);	Tests c to see if it is a printable ASCII character (including spaces)	8-70
int **ispunct**(int c);	Tests c to see if it is an ASCII punctuation character	8-70
int **isspace**(int c);	Tests c to see if it is an ASCII spacebar, tab (horizontal or vertical), carriage return, formfeed, or newline character	8-70
int **isupper**(int c);	Tests c to see if it is an uppercase alphabetic ASCII character	8-70
int **isxdigit**(int c);	Tests c to see if it is a hexadecimal digit	8-70
int **toascii**(int c);	Masks c into a legal ASCII value	8-107
int **tolower**(int c);	Converts c to lowercase if it is uppercase	8-108
int **toupper**(int c);	Converts c to uppercase if it is lowercase	8-108

Note: Functions in ctype.h/cctype are expanded inline if the –x option is used.

Table 8–3. Summary of Run-Time-Support Functions and Macros (Continued)

(c) Floating-point math functions (math.h/cmath)

Function	Description	Page
double **acos**(double x);	Returns the arc cosine of x	8-39
double **asin**(double x);	Returns the arc sine of x	8-42
double **atan**(double x);	Returns the arc tangent of x	8-44
double **atan2**(double y, double x);	Returns the inverse tangent of y/x	8-44
double **ceil**(double x);	Returns the smallest integer greater than or equal to x; expands to inline if –x option is used	8-48
double **cos**(double x);	Returns the cosine of x	8-50
double **cosh**(double x);	Returns the hyperbolic cosine of x	8-50
double **exp**(double x);	Returns the exponential function of x; expands inline unless –x0 option is used	8-53
double **fabs**(double x);	Returns the absolute value of x	8-54
double **floor**(double x);	Returns the largest integer less than or equal to x; expands inline if –x option is used	8-61
double **fmod**(double x, double y);	Returns the floating-point remainder of x/y; expands inline if –x option is used	8-61
double **frexp**(double value, int *exp);	Breaks value into a normalized fraction and an integer power of 2	8-65
double **ldexp**(double x, int exp);	Multiplies x by an integer power of 2	8-72
double **log**(double x);	Returns the natural logarithm of x	8-73
double **log10**(double x);	Returns the base-10 (common) logarithm of x	8-73
double **modf**(double value, double *iptr);	Breaks value into a signed integer and a signed fraction	8-79
double **pow**(double x, double y);	Returns x raised to the power y	8-80
double **sin**(double x);	Returns the sine of x	8-89
double **sinh**(double x);	Returns the hyperbolic sine of x	8-89
double **sqrt**(double x);	Returns the nonnegative square root of x	8-90
double **tan**(double x);	Returns the tangent of x	8-105
double **tanh**(double x);	Returns the hyperbolic tangent of x	8-106

Table 8–3. Summary of Run-Time-Support Functions and Macros (Continued)

(d) Nonlocal jumps macro and function (setjmp.h/csetjmp)

Macro or Function	Description	Page
int **setjmp**(jmp_buf env);	Saves calling environment for later use by longjmp function; expands inline if –x option is used	8-87
void **longjmp**(jmp_buf env, int _val);	Uses jmp_buf argument to restore a previously saved program environment	8-87

(e) Variable-argument functions and macros (stdarg.h/cstdarg)

Macro	Description	Page
type **va_arg**(_ap, type);	Accesses the next argument of type *type* in a variable-argument list	8-109
void **va_end**(_ap);	Resets the calling mechanism after using va_arg	8-109
void **va_start**(_ap, parmN);	Initializes ap to point to the first operand in the variable-argument list	8-109

(f) C/C++ I/O functions (stdio.h/cstdio)

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 table	8-40
void **clearerr**(FILE *_fp);	Clears the EOF and error indicators for the stream that _fp points to	8-49
int **fclose**(FILE *_fp);	Flushes the stream that _fp points to and closes the file associated with that stream	8-58
int **feof**(FILE *_fp);	Tests the EOF indicator for the stream that _fp points to	8-59
int **ferror**(FILE *_fp);	Tests the error indicator for the stream that _fp points to	8-59
int **fflush**(register FILE *_fp);	Flushes the I/O buffer for the stream that _fp points to	8-59
int **fgetc**(register FILE *_fp);	Reads the next character in the stream that _fp points to	8-60
int **fgetpos**(FILE *_fp, fpos_t *_pos);	Stores the object that _pos points to as the current value of the file position indicator for the stream that _fp points to	8-60

Table 8–3. Summary of Run-Time-Support Functions and Macros (Continued)

(f) C I/O functions (stdio.h/cstdio) (continued)

Function	Description	Page
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	8-60
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	8-62
int **fprintf**(FILE *_fp, const char *_format, ...);	Writes to the stream that _fp points to	8-62
int **fputc**(int _c, register FILE *_fp);	Writes a single character, _c, to the stream that _fp points to	8-62
int **fputs**(const char *_ptr, register FILE *_fp);	Writes the string pointed to by _ptr to the stream pointed to by _fp	8-63
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	8-63
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	8-65
int **fscanf**(FILE *_fp, const char *_fmt, ...);	Reads formatted input from the stream that _fp points to	8-66
int **fseek**(register FILE *_fp, long _offset, int _ptrname);	Sets the file position indicator for the stream that _fp points to	8-66
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.	8-67
long **ftell**(FILE *_fp);	Obtains the current value of the file position indicator for the stream that _fp points to	8-67
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	8-68
int **getc**(FILE *_p);	Reads the next character in the stream that _fp points to	8-68
int **getchar**(void);	A macro that calls fgetc() and supplies stdin as the argument	8-68
char ***gets**(char *_ptr);	Performs the same function as fgets() using stdin as the input stream	8-69
void **perror**(const char *_s);	Maps the error number in _s to a string and prints the error message	8-79
int **printf**(const char *_format, ...);	Performs the same function as fprintf() but uses stdout as its output stream	8-80

Table 8–3. Summary of Run-Time-Support Functions and Macros (Continued)

(f) C I/O functions (stdio.h/cstdio) (continued)

Function	Description	Page
int **putc**(int _x, FILE *_fp);	A macro that performs like fputc()	8-81
int **putchar**(int _x);	A macro that calls fputc() and uses stdout as the output stream	8-81
int **puts**(const char *_ptr);	Writes the string pointed to by _ptr to stdout	8-81
int **remove** (const char *_file);	Causes the file with the name pointed to by _file to be no longer available by that name	8-84
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	8-85
void **rewind** (register FILE *_fp);	Sets the file position indicator for the stream pointed to by _fp to the beginning of the file	8-85
int **scanf** (const char *_fmt, ...);	Performs the same function as fscanf() but reads input from stdin	8-86
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	8-86
int **setvbuf** (register FILE *_fp, register char *_buf, register int _type, register size_t _size);	Defines and associates a buffer with a stream	8-88
int **sprintf** (char *_string, const char *_format, ...);	Performs the same function as fprintf() but writes to the array that _string points to	8-90
int **sscanf** (const char *_str, const char *_fmt, ...);	Performs the same function as fscanf() but reads from the string that _str points to	8-91
FILE ***tmpfile**(void);	Creates a temporary file	8-107
char ***tmpnam**(char *_s);	Generates a string that is a valid filename (that is, the filename is not already being used)	8-107
int **ungetc** (int _c, register FILE *_fp);	Pushes the character specified by _c back into the input stream pointed to by _fp	8-108
int **vfprintf** (FILE *_fp, const char *_format, va_list _ap);	Performs the same function as fprintf() but replaces the argument list with _ap	8-110
int **vprintf** const char *_format, va_list _ap);	Performs the same function as printf() but replaces the argument list with _ap	8-110
int **vsprintf** (char *_string, const char *_format, va_list _ap);	Performs the same function as sprintf() but replaces the argument list with _ap	8-111

Table 8–3. Summary of Run-Time-Support Functions and Macros (Continued)

(g) General utilities (stdlib.h/cstdlib)

Function	Description	Page
void **abort**(void)	Terminates a program abnormally	8-38
int **abs**(int j);	Returns the absolute value of j; expands inline unless –x0 is used	8-39
void **atexit**(void (*fun)(void));	Registers the function pointed to by fun, called without arguments at normal program termination	8-45
double **atof**(const char *st);	Converts a string to a floating-point value; expands inline if –x is used	8-46
int **atoi**(register const char *st);	Converts a string to an integer value	8-46
long **atol**(register const char *st);	Converts a string to a long integer value; expands inline if –x is used	8-46
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	8-47
void ***calloc**(size_t num, size_t size);	Allocates and clears memory for num objects, each of size bytes	8-48
div_t **div**(register int numer, register int denom);	Divides numer by denom producing a quotient and a remainder	8-52
void **exit**(int status);	Terminates a program normally	8-53
void **free**(void *packet);	Deallocates memory space allocated by malloc, calloc, or realloc	8-64
int **free_memory**(void);	Returns the total dynamic memory available for al-loction	8-64
char ***getenv**(const char *_string)	Returns the environment information for the vari-able associated with _string	8-69
long **labs**(long i);	Returns the absolute value of i; expands inline unless –x0 is used	8-39
ldiv_t **ldiv**(long numer, long denom);	Divides numer by denom	8-72
int **ltoa**(long val, char *buffer);	Converts val to the equivalent string	8-74
void ***malloc**(size_t size);	Allocates memory for an object of size bytes	8-74
int **max_free**(void);	Returns the size of the biggest dynamic memory allocation possible	8-75
void **qsort**(void *_base, size_t nmemb, size_t size, int (*compar) (void));	Sorts an array of nmemb members; *base* points to the first member of the unsorted array, and *size* specifies the size of each member	8-82

Table 8–3. 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	8-83
void ***realloc**(void *packet, size_t size);	Changes the size of an allocated memory space	8-83
void **srand**(unsigned int seed);	Resets the random number generator	8-83
double **strtod**(const char *st, char **endptr);	Converts a string to a floating-point value	8-103
long **strtol**(const char *st, char **endptr, int base);	Converts a string to a long integer	8-103
unsigned long **strtoul**(const char *st, char **endptr**,** int base)**;**	Converts a string to an unsigned long integer	8-103

(h) String functions (string.h/cstring)

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 s; expands inline if –x is used	8-75
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	8-76
void ***memcpy**(void *s1, const void *s2, register size_t n);	Copies n characters from s2 to s1	8-76
void ***memmove**(void *s1, const void *s2, size_t n);	Moves n characters from s2 to s1	8-77
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 of –x is used	8-77
char ***strcat**(char *string1, const char *string2);	Appends string2 to the end of string1	8-91
char ***strchr**(const char *string, int c);	Finds the first occurrence of character c in s; expands inline if –x is used	8-91
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.	8-93
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	8-93
char ***strcpy**(register char *dest, register const char *src);	Copies string src into dest; expands inline if –x is used	8-94

Table 8–3. Summary of Run-Time-Support Functions and Macros (Continued)

(h) String functions (string.h/cstring) (continued)

Function	Description	Page
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	8-95
char ***strerror**(int errno);	Maps the error number in errno to an error message string	8-95
size_t **strlen**(char *string);	Returns the length of a string	8-97
char ***strncat**(char *dest, const char *src, register size_t n);	Appends up to n characters from src to dest	8-98
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	8-99
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	8-100
char ***strpbrk**(const char *string, const char *chs);	Locates the first occurrence in string of *any* character from chs	8-101
char ***strrchr**(const char *string, int c);	Finds the last occurrence of character c in string; expands inline if –x is used	8-101
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	8-102
char ***strstr**(register const char *string1, const char *string2);	Finds the first occurrence of string2 in string1	8-102
char ***strtok**(char *str1, const char *str2);	Breaks str1 into a series of tokens, each delimited by a character from str2	8-104
size_t **strxfrm**(register char *to, register const char *from, register size_t n);	Transforms n characters from *from*, to *to*	8-105

Table 8–3. Summary of Run-Time-Support Functions and Macros (Continued)

(i) Time functions (time.h/ctime)

Function	Description	Page
char ***asctime**(const struct tm *timeptr);	Converts a time to a string	8-42
clock_t **clock**(void);	Determines the processor time used	8-49
char ***ctime**(const time_t *timer);	Converts calendar time to local time	8-51
double **difftime**(time_t time1, time_t time0);	Returns the difference between two calendar times	8-51
struct tm ***gmtime**(const time_t *timer);	Converts calendar time to Greenwich Mean Time	8-70
struct tm ***localtime**(const time_t *timer);	Converts calendar time to local time	8-72
time_t **mktime**(register struct tm *tptr);	Converts local time to calendar time	8-78
size_t **strftime**(char *out, size_t maxsize, const char *format, const struct tm *time);	Formats a time into a character string	8-96
time_t **time**(time_t *timer);	Returns the current calendar time	8-106

(j) Summary of far versions

Function	Description	Page
int **far_atoi**(const far char *st)	Far version of function atoi()	8-46
long **far_atol**(const far char *st)	Far version of funciton atol()	8-46
double **far_atof**(const far char *st)	Far version of function atof()	8-46
far void ***far_bsearch**(const far void *key, const far void *base, size_t nmemb, size_t size, int (*compar) (const far void *, const far void *));	Far version of function bsearch()	8-47
far void ***far_calloc**(unsigned long num, unsigned long size)	Far version of function calloc()	8-48
void **far_free** (far void *ptr)	Far version of function free()	8-64
long **far_free_memory**(void)	Far version of function free_memory()	8-64
double **far_frexp**(double x, far int *exp)	Far version of function frexp()	8-65
void **far_longjmp**(far_jmp_buf env, int val);	Far version of function longjmp()	8-87
far void ***far_malloc**(unsigned long size)	Far version of function malloc()	8-74
long **far_max_free**(void)	Far version of function max_free()	8-75
far void ***far_memchr**(const far void *s, int c, size_t n);	Far version of function memchr()	8-75

Table 8–3. Summary of Run-Time-Support Functions and Macros (Continued)

(j) Summary of far versions (continued)

Function	Description	Page
int **far_memcmp**(const far void *s1, const far void *s2, size_t n);	Far version of function memcmp()	8-76
void far ***far_memcpy**(far void *_s1, far const void *_s2, size_t _n);	Far version of function memcpy()	8-76
far void ***far_memlcpy**(far void *to, const far void *from, unsigned long n);	Memory block copy of block > 64k words (non-overlapping)	8-57
far void ***far_memlmove**(far void *to, const far void *from, unsinged long n);	Memory block copy of block > 64k words (overlapping)	8-57
void far ***far_memmove**(far void *s1, far const void *s2, size_t n);	Far version of function memmove()	8-77
far void ***far_memset**(far void *s, int c, size_t n);	Far version of function memset()	8-77
double **far_modf**(double x, far double *y)	Far version of function modf()	8-79
void **far_qsort**(far void *base, size_t nmemb, size_t size, int (*compar) (const far void *, const far void *));	Far version of function qsort()	8-82
far void ***far_realloc**(far void *prt, unsigned long size)	Far version of function realloc()	8-83
int **far_setjmp**(far_jmp_buf env);	Far version of function setjmp()	8-87
far char ***far_strcat**(far char *s1, const far char *s2);	Far version of function strcat()	8-91
far char ***far_strchr**(const far char *s, int c);	Far version of function strchr()	8-92
int **far_strcmp**(const far char *s1, const far char *s2);	Far version of function strcmp()	8-93
int **far_strcoll**(const far char *s1, const far char *s2);	Far version of function strcoll()	8-93
char far ***far_strcpy**(char far *s1, far const char *s2, size_t n);	Far version of function strcpy()	8-94
size_t **far_strcspn**(const far char *s1, const far char *s2);	Far version of function strcspn()	8-95
far char ***far_strerror**(int errno);	Far version of function strerror()	8-95
size_t **far_strlen**(const far char *s);	Far version of function strlen()	8-97
far char ***far_strncat**(far char *s1, const far char *s2, size_t n);	Far version of function strncat()	8-98
int **far_strncmp**(const far char *s1, const far char *s2, size_t n);	Far version of function strncmp()	8-99

Table 8–3. Summary of Run-Time-Support Functions and Macros (Continued)

(j) Summary of far versions (continued)

Function	Description	Page
far char ***far_strncpy**(far char *s1, far const char *s2, size_t n);	Far version of function strncpy()	8-100
far char ***far_strpbrk**(const far char *s1, const far char *s2);	Far version of function strpbrk()	8-101
far char ***far_strrchr**(const far char *s, int c);	Far version of function strrchr()	8-101
size_t **far_strspn**(const far char *s1, const far char *s2);	Far version of function strspn()	8-102
far char ***far_strstr**(const far char *s1, const far char *s2);	Far version of function strstr()	8-102
double **far_strtod**(const far char *st, far char **endprt)	Far version of function strtod()	8-103
far char ***far_strtok**(far char *s1, const far char *s2);	Far version of function strtok()	8-104
long **far_strtol**(const far char *st, far char **endprt, int base)	Far version of function strtol()	8-103
unsigned long **far_strtoul**(const far char *st, far char **endprt, int base)	Far version of function strtoul()	8-103
size_t **far_strxfrm**(far char *s1, const far char *s2, size_t n);	Far version of function strxfrm()	8-105

8.6 Description of Run-Time-Support Functions and Macros

This section describes the runtime-support functions and macros. For each function or macro, the syntax is given in both C and C++. Because the functions and macros originated from the C header files, however, program examples are shown in C code only. The same program in C++ code would differ in that the types and functions declared in the header file are introduced into the std namespace.

abort — *Abort*

Syntax for C

#include <stdlib.h>

void abort(void**);**

Syntax for C++

#include <cstdlib>

void std::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 8-53.

Far Version None

abs/labs — *Absolute Value*

Syntax for C #include <stdlib.h>

int abs(int j**);**
long labs(long i**);**

Syntax for C++ #include <cstdlib>

int std::abs(int j**);**
long std::labs(long i**);**

Defined in abs.c in rts.src

Description The C/C++ compiler supports two functions that return the absolute value of an integer:

- ❑ The abs function returns the absolute value of an integer j.
- ❑ The labs function returns the absolute value of a long integer k.

Far Version None

acos — *Arc Cosine*

Syntax for C #include <math.h>

double acos(double x**);**

Syntax for C++ #include <cmath>

double std::acos(double x**);**

Defined in asin.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 (rrealval = 1.0;
radians = acos(realval);
return (radians);   /* acos return pi/2 */
```

Far Version None

add_device *Add Device to Device Table*

Syntax for C

```
#include <stdio.h>
int add_device(char *name,
                  unsigned flags,
                  int (*dopen)(), parameters,
                  int (*dclose)(), parameters,
                  int (*dread)(), parameters,
                  int (*dwrite)(), parameters,
                  fpos_t (*dlseek)(), parameters,
                  int (*dunlink)(), parameters,
                  int (*drename)(), parameters);
```

Syntax for C++

```
#include <cstdio>
int std::add_device(char *name,
                  unsigned flags,
                  int (*dopen)(), parameters,
                  int (*dclose)(), parameters,
                  int (*dread)(), parameters,
                  int (*dwrite)(), parameters,
                  fpos_t (*dlseek)(), parameters,
                  int (*dunlink)(), parameters,
                  int (*drename)(), parameters);
```

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 I/O from C/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 the device added.

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/cstdio.

- ❑ The dopen, dclose, dread, dwrite, dlseek, dunlink, and 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 8.3.1, *Overview of Low-Level I/O Implementation*, on page 8-9. The device drivers for the host that the debugger is run on are included in the C/C++ I/O library.

Return Value The function returns one of the following values:

0	if successful
–1	if fails

Example This example:

1) Adds the device *mydevice* to the device table

2) Opens a file named *test* on that device and associates it with the file **fid*

3) Prints the string *Hello, world* into the file

4) Closes the file

```
#include <stdio.h>

/****************************************************************************/
/* Declarations of the user–defined device drivers                          */
/****************************************************************************/
extern int my_open(const 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, const char *buffer, unsigned count);
extern int my_lseek(int fno, long offset, int origin);
extern int my_unlink(const char *path);
extern int my_rename(const char *old_name, const 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);
}
```

Far Version None

asctime — *Internal Time to String*

Syntax for C

#include <time.h>

char *asctime(const struct tm *timeptr**);**

Syntax for C++

#include <ctime>

char *std::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/ctime header declares and defines, see section 8.4.12, *Time Functions (time.h/ctime)*, on page 8-25.

Far Version

None

asin — *Arc Sine*

Syntax for C

#include <math.h>

double asin(double x**);**

Syntax for C++

#include <cmath>

double std::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 pi/2 */
```

Far Version

None

assert — *Insert Diagnostic Information*

Syntax for C

#include <assert.h>

void assert(int expr**);**

Syntax for C++

#include <cassert>

void assert(int expr**);**

Defined in

assert.h/cassert 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 and then aborts execution.
- ❑ If expr is true, the assert macro does nothing.

The header file that declares the assert macro refers to another macro, NDEBUG. If you have defined NDEBUG as a macro name when the assert.h header is included in the source file, 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;
```

Far Version

None

atan — *Polar Arc Tangent*

Syntax for C

#include <math.h>

double atan(double x**);**

Syntax for C++

#include <cmath>

double std::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 */
```

Far Version

None

atan2 — *Cartesian Arc Tangent*

Syntax for C

#include <math.h>

double atan2(double y, double x**);**

Syntax for C++

#include <cmath>

double std::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 */
```

Far Version

None

atexit — *Register Function Called by Exit ()*

Syntax for C #include <stdlib.h>

void atexit(void (*fun)(void)**);**

Syntax for C++ #include <cstdlib>

void std::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.

Far Version None

atof/atoi/atol *String to Number*

Syntax for C

#include <stdlib.h>

double atof(const char *st**);**
int atoi(const char *st**);**
long atol(const char *st**);**

Syntax for C++

#include <cstdlib>

double std::atof(const char *st**);**
int std::atoi(const char *st**);**
long std::atol(const char *st**);**

Defined in

atof.c, atoi.c, and atol.c in rts.src
faratof.c, faratoi,c, and faratol.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. In the atof stream, 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.

Because int and long are functionally equivalent in TMS320C28x C/C++, the atoi and atol functions are also functionally equivalent.

The functions do not handle any overflow resulting from the conversion.

Far Version (C)

double far_atof(const far char *st);
int far_atoi(const far char *st);
long far_atol(const far char *st);

bsearch *Array Search*

Syntax for C

#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 *)**);**

Syntax for C++

#include <cstdlib>

void *std::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 and farbsearch.c in rts.src

Description

The bsearch function searches through an array of nmemb objects for a member that matches the object that key points to. Argument base points to the first member in the array; size specifies the size (in bytes) of each member.

The contents of the array must be in ascending order. If a match is found, the function returns a pointer to the matching member of the array; if no match is found, the function returns a null pointer (0).

Argument compar points to a function that compares key to the array elements. The comparison function should be declared as:

```
int  cmp(const void *ptr1, const void *ptr2)
```

The cmp function compares the objects that prt1 and ptr2 point to and returns one of the following values:

< 0	if *ptr1 is less than *ptr2
0	if *ptr1 is equal to *ptr2
> 0	if *ptr1 is greater than *ptr2

Far Version (C)

far void *far_bsearch(const far void *key, const far void *base, size_t nmemb, size_t size, int (*compar) (const far void *, const far void *)**);**

calloc — *Allocate and Clear Memory*

Syntax for C #include <stdlib.h>

void *calloc(size_t num, size_t size**);**

Syntax for C++ #include <cstdlib>

void *std::calloc(size_t num, size_t size**);**

Defined in memory.c and farmemory.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 1K words. You can change this amount at link time by invoking the linker with the –heap option and specifying the desired size of the heap (in bytes) directly after the option. See section 7.1.4, *Dynamic Memory Allocation*, on page 7-7.

Example This example uses the calloc routine to allocate and clear 20 bytes.

```
prt = calloc (10,2)  ;   /*Allocate and clear 20 bytes */
```

Far Version (C) **far void *far_calloc(**unsigned long num, unsigned long size**);**

Far Version (C++) **long std::far_calloc(**unsigned long num, unsigned long size**);**

ceil — *Ceiling*

Syntax for C #include <math.h>

double ceil(double x**);**

Syntax for C++ #include <cmath>

double std::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 */
```

Far Version None

clearerr — *Clear EOF and Error Indicators*

Syntax for C

```
#include    <stdio.h>
```

void clearerr(FILE *_fp**);**

Syntax for C++

```
#include    <cstdio>
```

void std::clearerr(FILE *_fp**);**

Defined in clearerr.c in rts.src

Description The clearerr function clears the EOF and error indicators for the stream that _fp points to.

Far Version None

clock — *Processor Time*

Syntax for C

```
#include    <time.h>
```

clock_t clock(void**);**

Syntax for C++

```
#include    <ctime>
```

clock_t std::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. 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.

Far Version None

cos

Cosine

Syntax for C

#include <math.h>

double cos(double x**);**

Syntax for C++

#include <cmath>

double std::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 can 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 */
```

Far Version

None

cosh

Hyperbolic Cosine

Syntax for C

#include <math.h>

double cosh(double x**);**

Syntax for C++

#include <cmath>

double std::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 */
```

Far Version

None

ctime — *Calendar Time*

Syntax for C

#include <time.h>

char *ctime(const time_t *timer**);**

Syntax for C++

#include <ctime>

char *std::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/ctime header declares and defines, see section 8.4.12, *Time Functions (time.h/ctime)*, on page 8-25.

Far Version

None

difftime — *Time Difference*

Syntax for C

#include <time.h>

double difftime(time_t time1, time_t time0**);**

Syntax for C++

#include <ctime>

double std::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/ctime header declares and defines, see section 8.4.12, *Time Functions (time.h/ctime)*, on page 8-25.

Far Version

None

div/ldiv *Division*

Syntax for C

#include <stdlib.h>

div_t div(int numer, denom**);**
ldiv_t ldiv(long numer, denom**);**

Syntax for C++

#include <cstdlib>

div_t std::div(int numer, denom**);**
ldiv_t std::ldiv(long numer, denom**);**

Defined in

div.c in rts.src

Description

Two functions support integer division by returning numer (numerator) divided by denom (denominator). You can use these functions to get both the quotient and the remainder in a single operation.

- ❑ The div function performs integer division. The input arguments are integers; the function returns the quotient and the remainder in a structure of type div_t. The structure is defined as follows:

```
typedef struct
{
   int  quot;         /*  quotient    */
   int  rem;          /* remainder    */
} div_t;
```

- ❑ The ldiv function performs long integer division. The input arguments are long integers; the function returns the quotient and the remainder in a structure of type ldiv_t. The structure is defined as follows:

```
typedef struct
{
   long int  quot;    /*  quotient    */
   long int  rem;     /* remainder    */
} ldiv_t;
```

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.

Because ints and longs are equivalent types in TMS320C28x C/C++, ldiv and div are also equivalent.

Far Version

None

exit — *Normal Termination*

Syntax for C

#include <stdlib.h>

void exit(int status**);**

Syntax for C++

#include <cstdlib>

void std::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 8-38.)

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.

Far Version

None

exp — *Exponential*

Syntax for C

#include <math.h>

double exp(double x**);**

Syntax for C++

#include <cmath>

double std::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 */
```

Far Version

None

fabs — *Absolute Value*

Syntax for C

#include <math.h>

double fabs(double x**);**

Syntax for C++

#include <cmath>

double std::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 */
```

Far Version None

far_calloc — *Allocate and Clear Far Memory*

Syntax for C

#include <stdlib.h>
far void *far_calloc(unsigned long num, unsigned long size**);**

Syntax for C++

#include <scstdlib>
long std::far_calloc(unsigned long num, unsigned long size**);**

Defined in farmemory.c in rts.src

Description The far_calloc function allocates size bytes (size is an unsigned long) for each of num objects in the far heap 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 far_calloc uses is in a special memory pool or far heap. The constant _ _FAR_SYSMEM_SIZE defines the size of the heap as 1K words. You can change this amount at link time by invoking the linker with the –farheap option and specifying the desired size of the heap (in bytes) directly after the option. See section 7.1.4, *Dynamic Memory Allocation*, on page 7-7.

The C++ far_calloc function allocates space for an object in the far heap, clears the allocated memory and returns the address as long. You can use this address with the C++ far instrinsics.For more information, see section 6.7.8, Using Intrinsics to access Far Memory in C++, on page 6-19.

Example

The following C example uses far_calloc to allocate and clear 20 bytes in the far memory.

```
char *ptr = (char*) far_calloc(10,2);
/* allocate and clear 20 bytes */
```

The following is a C++ example.

```
long ptr = std::far_calloc(10,2);
/* allocate and clear 20 bytes */
```

far_free — *Deallocate Far Memory*

Syntax for C

#include <stdlib.h>
void far_free(far void *ptr**);**

Syntax for C++

#include <cstdlib>
void std::far_free(long ptr**);**

Defined in

farmemory.c in rts.src

Description

The far_free function deallocates far memory space (pointer to by ptr) that was previously allocated by a far_malloc, far_calloc, or far_realloc call. This makes the memory space available again. If you attempt to free unallocated space, the result is unpredictable. For more information, see section 7.1.4, *Dynamic Memory Allocation*, on page 7-7.

Example

The following C example allocates ten bytes and then frees them.

```
far char *x = (far char*) far_malloc(10);
/*allocate 10 bytes*/

far_free (x);
/*free 10 bytes*/
```

C++ example

```
long x = std::far_malloc(10);   /*allocate 10 bytes*/

std::far_free(x);               /*free 10 bytes*/
```

far_malloc — *Allocate Far Memory*

Syntax for C

#i#include <stdlib.h>
far void *far_malloc(unsigned long size**);**

Syntax for C++

#include <cstdlib>
long std::far_malloc(unsigned long size**);**

Defined in

farmemory.c in rts.src

Description

The far_malloc function allocates space for an object of size bytes in the far memory and returns a ponter to the space. If far_malloc cannot allocate the packet (that is, if it runs out of memory), it returns a null pointer (0). This function does not modify the memory it allocates.

The memory that far_malloc uses is in a special memory pool or heap. The constant _ _FAR_SYSMEM_SIZE defines the size of the heap as 1K words. You can change this amount at link time by invoking the linker with the –farheap option and specifying the desired size of the heap (in bytes) directly after the option. For more information, see section 7.1.4, *Dynamic Memory Allocation*, on page 7-7.

The C++ far_malloc function allocates space for an object in the far heap and returns the address as long. You can use this address with the C++ far intrinsics. For more information, see section 6.7.8, *Using Intrinsics to access Far Memory in C*++, on page 6-19.

Example

The following C code allocates 10 bytes of data in the far memory.

```
char *ptr = (char *) far_malloc(10);
/*allocate 10 bytes */
```

C++ example

```
long ptr = std:far_malloc(10);   /*allocate 10 bytes */
```

far_memlcpy — *Far Memory Block Copy of > 64K Block – Nonoverlapping*

Syntax for C

#include <string.h>
far void *far_memlcpy(far void *s1, const far void *s2, unsigned long n**);**

Syntax for C++

#inlcude <cstring>
long std::far_memlcpy(long s1, const long s2, unsigned long n**);**

Defined in

farmemory.c in rts.src

Description

The far_memlcpy function copies *n* characters from the object to which s2 points into the object to which s1 points. The number of characters to be copied can be more than 64K. If you attempt to copy characters of overlapping objects, the behavior of the function is undefined. The function returns the value of s1.

far_memlmove — *Far Memory Block Copy of > 64K Block – Overlapping*

Syntax for C

#include <string.h>
far void *far_memlmove(far void *s1, const far void *s2, unsigned long n**);**

Syntax for C++

#include <cstring>
long std::far_memlmove(long s1, const long s2, unsigned long n**);**

Defined in

farmemory.c in rts.src

Description

The far_memlmove function moves *n* characters from the object to which s2 points into the object to which s1 points. The function returns the value of s1. The number of characters to be copied can be more than 64K. The far_memlmove function correctly copies characters between overlapping objects.

far_realloc — *Change Heap Size*

Syntax for C

```
#include <stdlib.h>
void *far_realloc(void *packet, unsigned long size);
```

Syntax for C++

```
#include <cstdlib>
long std::far_realloc(long packet, unsigned long size);
```

Defined in far_memroy.c in rts.src

Description The far_realloc function changes the size of the allocated memory pointed to by packet to size bytes. The contents of the memory space (up to the lesser of the old and new sizes) is not changed.

- ❑ If packet is 0, far_realloc behaves like far_malloc
- ❑ if packet points to unallocated space, far_realloc takes no action and returns 0.
- ❑ If the space cannot be allocated, the original memory space is not changed, and far_realloc returns 0.
- ❑ If size = = 0 and packet is not null, far_realloc frees the space that packet points to.

If the entire object must be moved to allocate more space, far_realloc returns a pointer to the new space. Any memory freed by this operation is deallocated. If an error occurs, the function returns a null pointer (0).

The memory that far_realloc uses is a special memory pool or heap. The constant _ _FAR_SYSMEM_SIZE defines the size of the heap as 1K words. You can change this amount at link time by invoking the linker with the –farheap option and specifying the desired size of the heap (in bytes) directly after the option. For more information, see section 7.1.4, *Dynamic Memory Allocation*, on page 7-7.

fclose — *Close File*

Syntax for C

```
#include   <stdio.h>

int fclose(FILE *_fp);
```

Syntax for C++

```
#include   <cstdio>

int std::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.

Far Version None

feof — *Test EOF Indicator*

Syntax for C

#include <stdio.h>

int feof(FILE *_fp**);**

Syntax for C++

#include <cstdio>

int std::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.

Far Version None

ferror — *Test Error Indicator*

Syntax for C

#include <stdio.h>

int ferror(FILE *_fp**);**

Syntax for C++

#include <cstdio>

int std::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.

Far Version None

fflush — *Flush I/O Buffer*

Syntax for C

#include <stdio.h>

int fflush(register FILE *_fp**);**

Syntax for C++

#include <cstdio>

int std::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.

Far Version None

fgetc

Read Next Character

Syntax for C

```
#include <stdio.h>
```

int fgetc(register FILE *_fp**);**

Syntax for C++

```
#include <cstdio>
```

int std::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.

Far Version None

fgetpos

Store Object

Syntax for C

```
#include <stdio.h>
```

int fgetpos(FILE *_fp**,** fpos_t *pos**);**

Syntax for C++

```
#include <cstdio>
```

int std::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.

Far Version None

fgets

Read Next Characters

Syntax for C

```
#include <stdio.h>
```

char *fgets(char *_ptr, register int _size, register FILE *_fp**);**

Syntax for C++

```
#include <cstdio>
```

char *std::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.

Far Version None

floor *Floor*

Syntax for C

#include <math.h>

double floor(double x**);**

Syntax for C++

#include <cmath>

double std::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 */
```

Far Version

None

fmod *Floating-Point Remainder*

Syntax for C

#include <math.h>

double fmod(double x, double y**);**

Syntax for C++

#include <cmath>

double std::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 */
```

Far Version

None

fopen — *Open File*

Syntax for C

#include <stdio.h>

FILE ***fopen(**const char *_fname, const char *_mode**);**

Syntax for C++

#include <cstdio>

FILE ***std::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.

Far Version None

fprintf — *Write Stream*

Syntax for C

#include <stdio.h>

int fprintf(FILE *_fp, const char *_format, ...**);**

Syntax for C++

#include <cstdio>

int std::fprintf(FILE *_fp, const char *_format, ...**);**

Defined in fprintf.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.

Far Version None

fputc — *Write Character*

Syntax for C

#include <stdio.h>

int fputc(int _c, register FILE *_fp**);**

Syntax for C++

#include <cstdio>

int std::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.

Far Version None

fputs — *Write String*

Syntax for C #include <stdio.h>

int fputs(const char *_ptr, register FILE *_fp**);**

Syntax for C++ #include <cstdio>

int std::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.

Far Version None

fread — *Read Stream*

Syntax for C #include <stdio.h>

size fread(void *_ptr, size_t size, size_t count, FILE *_fp**);**

Syntax for C++ #include <cstdio>

size std::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.

Far Version None

free — *Deallocate Memory*

Syntax for C

#include <stdlib.h>

void free(void *ptr**);**

Syntax for C++

#include <cstdlib>

void std::free(void *ptr**);**

Defined in memory.c in rts.src

Description The free function deallocates memory space (pointed to by ptr) that was previously allocated by a malloc, calloc, or realloc call. This makes the memory space available again. If you attempt to free unallocated space, the result is undefined. For more information, see section 7.1.4, *Dynamic Memory Allocation*, on page 7-7.

Example This example allocates ten bytes and then frees them.

```
char *x;
x = malloc(10);       /*  allocate 10 bytes   */
free(x);              /*  free 10 bytes       */
```

Far Version (C) **void far_free(**void *ptr**);**

Far Version (C++) **void std::far_free(**long**)**

Refer to far_free located on page 8-55.

free_memory — *Returns the Total Dynamic Memory Available for Allocation*

Syntax for C

#include <stdlib.h>
int free_memory(void**);**

Syntax for C++

#include <cstdlib>
int std:free_memory (void**);**

Defined in memory.c and farmemory.c in rts.src

Description Call this function to find the total dynamic memory available for allocation. This function returns 0 if no memory is available.

Far Version (C) **long far_free_memory(**void**);**

Far Version (C++) **long std::far_free_memory(**void**);**

freopen — *Open File*

Syntax for C

#include <stdio.h>

FILE ***freopen(**const char *_fname, const char *_mode, register FILE *_fp**);**

Syntax for C++

#include <cstdio>

FILE ***std::freopen(**const char *_fname, const char *_mode,
register FILE *_fp**);**

Defined in

freopen.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.

Far Version

None

frexp — *Fraction and Exponent*

Syntax for C

#include <math.h>

double frexp(double value**,** int *exp**)**;

Syntax for C++

#include <cmath>

double std::frexp(double value**,** int *exp**)**;

Defined in

frexp.c andfarfrexp.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 */
```

Far Version

double far_frexp(double value, far int *exp**);**

fscanf *Read Stream*

Syntax for C

#include <stdio.h>

int fscanf(FILE *_fp, const char *_fmt, ...**);**

Syntax for C++

#include <cstdio>

int std::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.

Far Version None

fseek *Set File Position Indicator*

Syntax for C

#include <stdio.h>

int fseek(register FILE *_fp, long _offset, int _ptrname**);**

Syntax for C++

#include <cstdio>

int std::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.

Far Version None

fsetpos

Set File Position Indicator

Syntax for C	#include <stdio.h>
	int fsetpos(FILE *_fp, const fpos_t *_pos**);**
Syntax for C++	#include <cstdio>
	int std::fsetpos(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.
Far Version	None

ftell

Get Current File Position Indicator

Syntax for C	#include <stdio.h>
	long ftell(FILE *_fp**);**
Syntax for C++	#include <cstdio>
	long std::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.
Far Version	None

fwrite — *Write Block of Data*

Syntax for C

#include <stdio.h>

size_t fwrite(const void *_ptr, size_t _size, size_t _count, register FILE ***_fp**);**

Syntax for C++

#include <cstdio>

size_t std::fwrite(const void *_ptr, size_t _size, size_t _count,
register FILE ***_fp**);**

Defined in fwrite.c in rts.src

Description The fwrite function writes a block of data from the memory pointed to by _ptr to the stream that _fp points to.

Far Version None

getc — *Read Next Character*

Syntax for C

#include <stdio.h>

int getc(FILE *_fp**);**

Syntax for C++

#include <cstdio>

int std::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.

Far Version None

getchar — *Read Next Character From Standard Input*

Syntax for C

#include <stdio.h>

int getchar(void**);**

Syntax for C++

#include <cstdio>

int std::getchar(void**);**

Defined in fgetc.c in rts.src

Description The getchar function reads the next character from the standard input device.

Far Version None

getenv

Get Environment Information

Syntax for C

#include <stdlib.h>

char *getenv(const char *_string**);**

Syntax for C++

#include <cstdlib>

char *std::getenv(const char *_string**);**

Defined in

fgetenv.c in rts.src

Description

The getenv function returns the environment information for the variable associated with _string.

Note: The getenv Function Is Target-System Specific

The getenv function is target-system specific, so you must write your own getenv function.

Far Version

None

gets

Read Next From Standard Input

Syntax for C

#include <stdio.h>

char *gets(char *_ptr**);**

Syntax for C++

#include <cstdio>

char *std::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.

Far Version

None

gmtime — *Greenwich Mean Time*

Syntax for C

```
#include   <time.h>
```

struct tm *gmtime(const time_t *timer**);**

Syntax for C++

```
#include   <ctime>
```

struct tm *std::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/ctime header declares and defines, see section 8.4.12, *Time Functions (time.h/ctime)*, on page 8-25.

Far Version None

isxxx — *Character Typing*

Syntax for C

```
#include   <ctype.h>

int isalnum(int c);          int islower(int c);
int isalpha(int c);          int isprint(int c);
int isascii(int c);          int ispunct(int c);
int iscntrl(int c);          int isspace(int c);
int isdigit(int c);          int isupper(int c);
int isgraph(int c);          int isxdigit(int c);
```

Syntax for C++

```
#include   <cctype>

int std::isalnum(int c);     int std::islower(int c);
int std::isalpha(int c);     int std::isprint(int c);
int std::isascii(int c);     int std::ispunct(int c);
int std::iscntrl(int c);     int std::isspace(int c);
int std::isdigit(int c);     int std::isupper(int c);
int std::isgraph(int c);     int std::isxdigit(int c);
```

Defined in isxxx.c and ctype.c in rts.src
Also defined in ctype.h as macros

Description These functions test a single argument c to see if it is a particular type of character — alphabetic, alphanumeric, numeric, ASCII, etc. If the test is true, the function returns a nonzero value; if the test is false, the function returns 0. 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/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.

Far Version None

labs

See abs/labs on page 8-39.

ldexp — *Multiply by a Power of 2*

Syntax for C #include <math.h>

double ldexp(double x**,** int exp**);**

Syntax for C++ #include <cmath>

double std::ldexp(double x**,** int exp**);**

Defined in ldexp.c in rts.src

Description The ldexp function multiplies a floating-point number by the power of 2 and returns $x \times 2^{exp}$. The exp can be a negative or a positive value. A range error occurs if the result is too large.

Example

```
double result;

result = ldexp(1.5, 5);         /* result is 48.0 */
result = ldexp(6.0, -3);        /* result is 0.75 */
```

Far Version None

ldiv — *See div/ldiv on page 8-52.*

localtime — *Local Time*

Syntax for C #include <time.h>

struct tm *localtime(const time_t *timer**);**

Syntax for C++ #include <ctime>

struct tm *std::localtime(const time_t *timer**);**

Defined in localtime.c in rts.src

Description The localtime function converts a calendar time (pointed to by timer) into a broken-down time, which is expressed as local time. The function returns a pointer to the converted time.

For more information about the functions and types that the time.h/ctime header declares and defines, see section 8.4.12, *Time Functions (time.h/ctime)*, on page 8-25.

Far Version None

log — *Natural Logarithm*

Syntax for C

#include <math.h>

double log(double x**);**

Syntax for C++

#include <cmath>

double std::log(double x**);**

Defined in

log.c in rts.src

Description

The log function returns the natural logarithm of a real number x. A domain error occurs if x is negative; a range error occurs if x is 0.

Description

```
float x, y;

x = 2.718282;
y = log(x);        /* Return value = 1.0 */
```

Far Version

None

log10 — *Common Logarithm*

Syntax for C

#include <math.h>

double log10(double x**);**

Syntax for C++

#include <cmath>

double std::log10(double x**);**

Defined in

log10.c in rts.src

Description

The log10 function returns the base-10 logarithm of a real number x. A domain error occurs if x is negative; a range error occurs if x is 0.

Example

```
float x, y;

x = 10.0;
y = log(x);        /* Return value = 1.0 */
```

Far Version

None

longjmp — *See setjmp/longjmp on page 8-87.*

ltoa — *Long Integer to ASCII*

Syntax for C no prototype provided

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

Syntax for C++ no prototype provided

int std::ltoa(long val, char *buffer**);**

Defined in ltoa.c and farltoa.c in rts.src

Description The ltoa function is a nonstandard (non-ANSI) function and is provided for compatibility with non-ANSI code. The standard equivalent is sprintf. The function is not prototyped in rts.src. The ltoa function converts a long integer n to an equivalent ASCII string and writes it into the buffer. If the input number val is negative, a leading minus sign is output. The ltoa function returns the number of characters placed in the buffer.

Far Version **int far_ltoa(**long val, far char *buffer**);**

malloc — *Allocate Memory*

Syntax for C #include <stdlib.h>

void *malloc(size_t size**);**

Syntax for C++ #include <cstdlib>

void *std::malloc(size_t size**);**

Defined in memory.c in rts.src

Description The malloc function allocates space for an object of size bytes and returns a pointer to the space. If malloc cannot allocate the packet (that is, if it runs out of memory), it returns a null pointer (0). This function does not modify the memory it allocates.

The memory that malloc uses is in a special memory pool or heap. The constant _ _SYSMEM_SIZE defines the size of the heap as 1K words. You can change this amount at link time by invoking the linker with the –heap option and specifying the desired size of the heap (in bytes) directly after the option. For more information, see section 7.1.4, *Dynamic Memory Allocation*, on page 7-7.

Far Version (C) **far void *far_malloc(**unsigned long size**);**

Far Version (C++) **long std::far_malloc(**unsigned long size**);**

Refer to far_malloc located on page 8-56.

max_free

Returns the size of the biggest dynamic allocation possible

Syntax for C

#include <stdlib.h>
int max_free(void**);**

Syntax for C++

#include <cstdlib>
int std::max_free(void**);**

Defined in

memory.c and farmemory.c in rts.src and memory.cpp

Description

Returns the size of biggest contiguous memory block available for allocation using malloc, calloc, or realloc. This function returns 0 if no memory is available.

Far Version (C)

long far_max_free(void**);**

Far Version (C++)

long std::far_max_free(void**);**

memchr

Find First Occurrence of Byte

Syntax for C

#include <string.h>

void *memchr(const void *cs, int c, size_t n**);**

Syntax for C++

#include <cstring>

void *std::memchr(const void *cs, int c, size_t n**);**

Defined in

memchr.c and farmemchr.c in rts.src

Description

The memchr function finds the first occurrence of c in the first n characters of the object that cs points to. If the character is found, memchr returns a pointer to the located character; otherwise, it returns a null pointer (0).

The memchr function is similar to strchr, except that the object that memchr searches can contain values of 0 and c can be 0.

Far Version (C)

void *far_memchr(const far void *cs, int c, size_t n**);**

memcmp — *Memory Compare*

Syntax for C

#include <string.h>

int memcmp(const void *cs, const void *ct, size_t n**);**

Syntax for C++

#include <cstring>

int std::memcmp(const void *cs, const void *ct, size_t n**);**

Defined in

memcmp.c and farmemcmp.c in rts.src

Description

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

< 0	if *cs is less than *ct
0	if *cs is equal to *ct
> 0	if *cs is greater than *ct

The memcmp function is similar to strncmp, except that the objects that memcmp compares can contain values of 0.

Far Version (C)

int far_memcmp(const far void *cs, const far void *ct, size_t n**);**

memcpy — *Memory Block Copy — Nonoverlapping*

Syntax for C

#include <string.h>

void *memcpy(void *s1, const void *s2, size_t n**);**

Syntax for C++

#include <cstring>

void *std::memcpy(void *s1, const void *s2, size_t n**);**

Defined in

memcpy.c and farmemcpy.c in rts.src

Description

The memcpy function copies n characters from the object that s2 points to into the object that s1 points to. If you attempt to copy characters of overlapping objects, the function's behavior is undefined. The function returns the value of s1.

The memcpy function is similar to strncpy, except that the objects that memcpy copies can contain values of 0.

Far Version (C)

far void *far_memcpy(far void *s1; const far foid *s2, size_t n**);**

memmove — *Memory Block Copy — Overlapping*

Syntax for C

#include <string.h>

void *memmove(void *s1, const void *s2, size_t n**);**

Syntax for C++

#include <cstring>

void *std::memmove(void *s1, const void *s2, size_t n**);**

Defined in

memmove.c and farmemmove.c in rts.src

Description

The memmove function moves n characters from the object that s2 points to into the object that s1 points to; the function returns the value of s1. The memmove function correctly copies characters between overlapping objects.

Far Version (C)

far void *far_memmove(far void *s1, const far void *s2, size_t n**);**

memset — *Duplicate Value in Memory*

Syntax for C

#include <string.h>

void *memset(void *mem, register int ch, size_t length**);**

Syntax for C++

#include <cstring>

void *std::memset(void *mem, register int ch, size_t length**);**

Defined in

memset.c and farmemset.c in rts.src

Description

The memset function copies the value of ch into the first length characters of the object that mem points to. The function returns the value of mem.

Far Version (C)

far void *far_memset(far void *mem, register int ch, size_t length**);**

mktime — *Convert to Calendar Time*

Syntax for C

#include <time.h>

time_t *mktime(struct tm *timeptr**);**

Syntax for C++

#include <ctime>

time_t *std::mktime(struct tm *timeptr**);**

Defined in

mktime.c in rts.src

Description

The mktime function converts a broken-down time, expressed as local time, into proper calendar time. The timeptr argument points to a structure that holds the broken-down time.

The function ignores the original values of tm_wday and tm_yday and does not restrict the other values in the structure. After successful completion of time conversions, tm_wday and tm_yday are set appropriately, and the other components in the structure have values within the restricted ranges. The final value of tm_mday is not sent until tm_mon and tm_year are determined.

The return value is encoded as a value of type time_t. If the calendar time cannot be represented, the function returns the value –1.

For more information about the functions and types that the time.h/ctime header declares and defines, see section 8.4.12, *Time Functions (time.h/ctime)*, on page 8-25.

Example

This example determines the day of the week that July 4, 2001, falls on.

```
#include <time.h>
static const char *const wday[] = {
            ”Sunday”, ”Monday”, ”Tuesday”, ”Wednesday”,
            ”Thursday”, ”Friday”, ”Saturday” };

struct tm time_str;

time_str.tm_year  = 2001 – 1900;
time_str.tm_mon   = 7;
time_str.tm_mday  = 4;
time_str.tm_hour  = 0;
time_str.tm_min   = 0;
time_str.tm_sec   = 1;
time_str.tm_isdst = 1;

mktime(&time_str);

/* After calling this function, time_str.tm_wday    */
/*    contains the day of the week for July 4, 2001 */
```

Far Version

None

modf — *Signed Integer and Fraction*

Syntax for C

#include <math.h>

double modf(double value**,** double *iptr**);**

Syntax for C++

#include <cmath>

double std::modf(double value**,** double *iptr**);**

Defined in

modf.c and farmodf.cin rts.src

Description

The modf function breaks a value into a signed integer and a signed fraction. Each of the two parts has the same sign as the input argument. The function returns the fractional part of value and stores the integer as a double at the object pointed to by iptr.

Example

```
double value, ipart, fpart;

value = –3.1415;

fpart = modf(value, &ipart);


/* After execution, ipart contains –3.0, */
/* and fpart contains –0.1415.           */
```

Far Version (C)

double far_modf(double value, far double *iptr**);**

perror — *Map Error Number*

Syntax for C

#include <stdio.h>

void perror(const char *_s**);**

Syntax for C++

#include <cstdio>

void std::perror(const char *_s**);**

Defined in

perror.c in rts.src

Description

The perror function maps the error number in s to a string and prints the error message.

Far Version

None

pow — *Raise to a Power*

Syntax for C

#include <math.h>

double pow(double x, double y**);**

Syntax for C++

#include <cmath>

double std::pow(double x, double y**);**

Defined in pow.c in rts.src

Description The pow function returns x raised to the power y. A domain error occurs if $x = 0$ and $y \leq 0$, or if x is negative and y is not an integer. A range error occurs if the result is too large to represent.

Example

```
double x, y, z;

x = 2.0;
y = 3.0;
x = pow(x, y);        /* return value = 8.0 */
```

Far Version None

printf — *Write to Standard Output*

Syntax for C

#include <stdio.h>

int printf(const char *_format, ...**);**

Syntax for C++

#include <cstdio>

int std::printf(const char *_format, ...**);**

Defined in printf.c in rts.src

Description The printf function writes to the standard output device. The string pointed to by _format describes how to write the stream.

Far Version None

putc

Write Character

Syntax for C

#include <stdio.h>

int putc(int _x, FILE *_fp**);**

Syntax for C++

#include <cstdlib>

int std::putc(int _x, FILE *_fp**);**

Defined in putc.c in rts.src

Description The putc function writes a character to the stream pointed to by _fp.

Far Version None

putchar

Write Character to Standard Output

Syntax for C

#include <stdlib.h>

int putchar(int _x**);**

Syntax for C++

#include <cstdlib>

int std::putchar(int _x**);**

Defined in putchar.c in rts.src

Description The putchar function writes a character to the standard output device.

Far Version None

puts

Write to Standard Output

Syntax for C

#include <stdlib.h>

int puts(const char *_ptr**);**

Syntax for C++

#include <cstdlib>

int std::puts(const char *_ptr**);**

Defined in puts.c in rts.src

Description The puts function writes the string pointed to by _ptr to the standard output device.

Far Version None

qsort *Array Sort*

Syntax for C

#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size, int (*compar) ()**);**

Syntax for C++

#include <cstdlib>

void std::qsort(void *base, size_t nmemb, size_t size, int (*compar) ()**);**

Defined in

qsort.c and farqsort.c in rts.src

Description

The qsort function sorts an array of nmemb members. Argument base points to the first member of the unsorted array; argument size specifies the size of each member.

This function sorts the array in ascending order.

Argument compar points to a function that compares key to the array elements. The comparison function should be declared as:

```
int  cmp(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

Far Version (C)

void far_qsort(far void *base, size_t nmemb, size_t size, int (*compar) ()**);**

rand/srand

Random Integer

Syntax for C

#include <stdlib.h>

int rand(void**);**
void srand(unsigned int seed**);**

Syntax for C++

#include <cstdlib>

int std::rand(void**);**
void std::srand(unsigned int seed**);**

Defined in

rand.c in rts.src

Description

Two functions work together to provide pseudorandom sequence generation:

- ❑ The rand function returns pseudorandom integers in the range 0−RAND_MAX.
- ❑ The srand function sets the value of seed so that a subsequent call to the rand function produces a new sequence of pseudorandom numbers. The srand function does not return a value.

If you call rand before calling srand, rand generates the same sequence it would produce if you first called srand with a seed value of 1. If you call srand with the same seed value, rand generates the same sequence of numbers.

Far Version

None

realloc

Change Heap Size

Syntax for C

#include <stdlib.h>

void *realloc(void *packet, size_t size**);**

Syntax for C++

#include <cstdlib>

void *std::realloc(void *packet, size_t size**);**

Defined in

memory.c in rts.src

Description

The realloc function changes the size of the allocated memory pointed to by packet to the size specified in bytes by size. The contents of the memory space (up to the lesser of the old and new sizes) is not changed.

- ❑ If packet is 0, realloc behaves like malloc.
- ❑ If packet points to unallocated space, realloc takes no action and returns 0.

- ❑ If the space cannot be allocated, the original memory space is not changed, and realloc returns 0.
- ❑ If size = = 0 and packet is not null, realloc frees the space that packet points to.

If the entire object must be moved to allocate more space, realloc returns a pointer to the new space. Any memory freed by this operation is deallocated. If an error occurs, the function returns a null pointer (0).

The memory that realloc uses is in a special memory pool or heap. The constant _ _SYSMEM_SIZE defines the size of the heap as 1K words. You can change this amount at link time by invoking the linker with the –heap option and specifying the desired size of the heap (in bytes) directly after the option. For more information, see section 7.1.4, *Dynamic Memory Allocation*, on page 7-7.

Far Version (C) **far void *far_realloc(**far void *packet, size_t size**);**

Far Version (C++) **long std::far_realloc(**long packet, unsigned long size**);**

Refer to far_realloc located on page 8-58.

remove — *Remove File*

Syntax for C

#include <stdlib.h>

int remove(const char *_file**);**

Syntax for C++

#include <cstdlib>

int std::remove(const char *_file**);**

Defined in remove.c in rts.src

Description The remove function makes the file pointed to by _file no longer available by that name.

Far Version None

rename — *Rename File*

Syntax for C	#include <stdlib.h>
	int rename(const char *old_name, const char *new_name**);**
Syntax for C++	#include <cstdlib>
	int std::rename(const char *old_name, const char *new_name**);**
Defined in	rename.c in rts.src
Description	The rename function renames the file pointed to by old_name. The new name is pointed to by new_name.
Far Version	None

rewind — *Position File-Position Indicator to Beginning of File*

Syntax for C	#include <stdlib.h>
	void rewind(register FILE *_fp**);**
Syntax for C++	#include <cstdlib>
	void std::rewind(register FILE *_fp**);**
Defined in	rewind.c in rts.src
Description	The rewind function sets the file position indicator for the stream pointed to by _fp to the beginning of the file.
Far Version	None

scanf *Read Stream From Standard Input*

Syntax for C

#include <stdlib.h>

int scanf(const char *_fmt, ...**);**

Syntax for C++

#include <cstdlib>

int std::scanf(const char *_fmt, ...**);**

Defined in fscanf.c in rts.src

Description The scanf function reads from the stream from the standard input device. The string pointed to by _fmt describes how to read the stream.

Far Version None

setbuf *Specify Buffer for Stream*

Syntax for C

#include <stdlib.h>

void setbuf(register FILE *_fp, char *_buf**);**

Syntax for C++

#include <cstdlib>

void std::setbuf(register FILE *_fp, char *_buf**);**

Defined in setbuf.c in rts.src

Description The setbuf function specifies the buffer used by the stream pointed to by _fp. If _buf is set to null, buffering is turned off. No value is returned.

Far Version None

setjmp/longjmp *Nonlocal Jumps*

Syntax for C

```
#include <setjmp.h>

int setjmp(jmp_buf env)
void longjmp(jmp_buf env, int _val)
```

Syntax for C++

```
#include <csetjmp>

int std::setjmp(jmp_buf env)
void std::longjmp(jmp_buf env, int _val)
```

Defined in setjmp16.asm and setjmp32.asm in rts.src

Description The setjmp.h header defines one type, one macro, and one function for bypassing the normal function call and return discipline:

- The **jmp_buf** type is an array type suitable for holding the information needed to restore a calling environment.
- The **setjmp** macro saves its calling environment in the jmp_buf argument for later use by the longjmp function.

  If the return is from a direct invocation, the setjmp macro returns the value 0. If the return is from a call to the longjmp function, the setjmp macro returns a nonzero value.
- The **longjmp** function restores the environment that was saved in the jmp_buf argument by the most recent invocation of the setjmp macro. If the setjmp macro was not invoked, or if it terminated execution irregularly, the behavior of longjmp is undefined.

  After longjmp is completed, the program execution continues as if the corresponding invocation of setjmp had just returned the value specified by _val. The longjmp function does not cause setjmp to return a value of 0, even if _val is 0. If _val is 0, the setjmp macro returns the value 1.

The setjmp.h also defines the type far_jmp_buf, the far version of jmp_buf.

Example

These functions are typically used to effect an immediate return from a deeply nested function call:

```
#include <setjmp.h>

jmp_buf env;

main()
{
   int errcode;

   if ((errcode = setjmp(env)) == 0)
      nest1();
   else
      switch (errcode)
   . . .
}
   . . .
nest42()
{
   if (input() == ERRCODE42)
   /* return to setjmp call in main */
      longjmp (env, ERRCODE42);
   . . .
}
```

Far Version (C)

int far_setjmp(far_jmp_buf env**);**
void far_longjmp(far_jmp_buf env, int val**);**

setvbuf — *Define and Associate Buffer With Stream*

Syntax for C

#include <stdlib.h>

int **setvbuf(**register FILE *_fp, register char *_buf, register int _type,
register size_t _size**);**

Syntax for C++

#include <cstdlib>

int **std::setvbuf(**register FILE *_fp, register char *_buf, register int _type,
register size_t _size**);**

Defined in

setvbuf.c in rts.src

Description

The setvbuf function defines and associates the buffer used by the stream pointed to by _fp. If _buf is set to null, a buffer is allocated. If _buf names a buffer, that buffer is used for the stream. The _size specifies the size of the buffer. The _type specifies the type of buffering as follows:

_IOFBF	Full buffering occurs
_IOLBF	Line buffering occurs
_IONBF	No buffering occurs

Far Version

None

sin — *Sine*

Syntax for C

#include <math.h>

double sin(double x**);**

Syntax for C++

#include <cmath>

double std::sin(double x**);**

Defined in sin.c in rts.src

Description The sin function returns the sine of a floating-point number x. The angle x is expressed in radians. An argument with a large magnitude may produce a result with little or no significance.

Example

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

radian = 3.1415927;
sval = sin(radian);       /* –1 is returned by sin  */
```

Far Version None

sinh — *Hyperbolic Sine*

Syntax for C

#include <math.h>

double sinh(double x**);**

Syntax for C++

#include <cmath>

double std::sinh(double x**);**

Defined in sinh.c in rts.src

Description The sinh function returns the hyperbolic sine of a floating-point number x. A range error occurs if the magnitude of the argument is too large.

Example

```
double x, y;

x = 0.0;
y = sinh(x);      /* return value = 0.0 */
```

Far Version None

sprintf — *Write Stream*

Syntax for C

#include <stdlib.h>

int sprintf(char _string, const char *_format, ...**);**

Syntax for C++

#include <cstdlib>

int std::sprintf(char _string, const char *_format, ...**);**

Defined in sprintf.c in rts.src

Description The sprintf function writes to the array pointed to by _string. The string pointed to by _format describes how to write the stream.

Far Version None

sqrt — *Square Root*

Syntax for C

#include <math.h>

double sqrt(double x**);**

Syntax for C++

#include <cmath>

double std::sqrt(double x**);**

Defined in sqrt.c in rts.src

Description The sqrt function returns the nonnegative square root of a real number x. A domain error occurs if the argument is negative.

Example

```
double x, y;

x = 100.0;
y = sqrt(x);      /* return value = 10.0 */
```

Far Version None

srand — *See rand/srand on page 8-83.*

sscanf — *Read Stream*

Syntax for C

#include <stdlib.h>

int sscanf(const char *str, const char *format, ...**);**

Syntax for C++

#include <cstdlib>

int std::sscanf(const char *str, const char *format, ...**);**

Defined in

sscanf.c in rts.src

Description

The sscanf function reads from the string pointed to by str. The string pointed to by _format describes how to read the stream.

Far Version

None

strcat — *Concatenate Strings*

Syntax for C

#include <string.h>

char *strcat(char *string1, char *string2**);**

Syntax for C++

#include <cstring>

char *std::strcat(char *string1, char *string2**);**

Defined in

strcat.c and farstrcat.c in rts.src

Description

The strcat function appends a copy of string2 (including a terminating null character) to the end of string1. The initial character of string2 overwrites the null character that originally terminated string1. The function returns the value of string1.

Example In the following example, the character strings pointed to by *a, *b, and *c were assigned to point to the strings shown in the comments. In the comments, the notation \0 represents the null character:

```
char *a, *b, *c;
.
.
.
/* a --> ”The quick black fox\0”                              */
/* b --> ” jumps over \0”                                     */
/* c --> ”the lazy dog.\0”                                    */
strcat (a,b);
/* a --> ”The quick black fox jumps over \0”                  */
/* b --> ” jumps over \0”                                     */
/* c --> ”the lazy dog.\0” */
strcat (a,c);
/*a --> ”The quick black fox jumps over the lazy dog.\0”*/
/* b --> ” jumps over \0”                                     */
/* c --> ”the lazy dog.\0”                                    */
```

Far Version (C) **far char *far_strcat(**far char *string1, far char *string2**);**

strchr — *Find First Occurrence of a Character*

Syntax for C

#include <string.h>

char *strchr(const char *string, int c**);**

Syntax for C++

#include <cstring>

char *std::strchr(const char *string, int c**);**

Defined in strchr.c and farstrchr.c in rts.src

Description The strchr function finds the first occurrence of c in string. If strchr finds the character, it returns a pointer to the character; otherwise, it returns a null pointer (0).

Example

```
char *a = ”When zz comes home, the search is on for zs.”;
char *b;
char the_z = ’z’;
b = strchr(a,the_z);
```

After this example, *b points to the first z in zz.

Far Version (C) **far char *far_strchr(**const far char *string, int c**);**

strcmp/strcoll — *String Compare*

Syntax for C

#include <string.h>

int strcmp(const char *string1, const char *string2**);**
int strcoll(const char *string1, const char *string2**);**

Syntax for C++

#include <cstring>

int std::strcmp(const char *string1, const char *string2**);**
int std::strcoll(const char *string1, const char *string2**);**

Defined in

strcmp.c and farstrcmp.c in rts.src

Description

The strcmp and strcoll functions compare string2 with string1. The functions are equivalent; both functions are supported to provide compatibility with ANSI C/C++.

The functions return one of the following values:

< 0	if *string1 is less than *string2
0	if *string1 is equal to *string2
> 0	if *string1 is greater than *string2

Example

```
char *stra = ”why ask why”;
char *strb = ”just do it”;
char *strc = ”why ask why”;

if (strcmp(stra, strb) > 0)
   {
      /*    statements here will be executed    */
   }
if (strcoll(stra, strc) == 0)
   {
      /* statements here will be executed also */
   }
```

Far Version (C)

int far_strcmp(const far char *string1, const far char *string2**);**
int far_strcoll(const far char *string1, const far char *string2**);**

strcpy — *String Copy*

Syntax for C

#include <string.h>

char *strcpy(char *dest, register const char *src**);**

Syntax for C++

#include <cstring>

char *std::strcpy(char *dest, register const char *src**);**

Defined in

strcpy.c and farstrcpy.c in rts.src

Description

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

Example

In the following example, the strings pointed to by *a and *b are two separate and distinct memory locations. In the comments, the notation \0 represents the null character:

```
char *a = ”The quick black fox”;
char *b = ” jumps over ”;

/* a ––> ”The quick black fox\0”   */
/* b ––> ” jumps over \0”          */

strcpy(a,b);

/* a ––> ” jumps over \0”          */
/* b ––> ” jumps over \0”          */
```

Far Version (C)

far char *far_strcpy(far char *dest, const far char *src**);**

strcspn — *Find Number of Unmatching Characters*

Syntax for C

#include <string.h>

size_t strcspn(const char *string, const char *chs**);**

Syntax for C++

#include <cstring>

size_t std::strcspn(const char *string, const char *chs**);**

Defined in

strcspn.c and farstrcspn.c in rts.src

Description

The strcspn function returns the length of the initial segment of string, which is made up entirely of characters that are not in chs. If the first character in string is in chs, the function returns 0.

Example

```
char *stra = ”who is there?”;
char *strb = ”abcdefghijklmnopqrstuvwxyz”;
char *strc = ”abcdefg”;
size_t length;

length = strcspn(stra,strb);    /* length = 0 */
length = strcspn(stra,strc);    /* length = 9 */
```

Far Version (C)

size_t far_strcspn(const far char *string, const far char *chs**);**

strerror — *String Error*

Syntax for C

#include <string.h>

char *strerror(int errno**);**

Syntax for C++

#include <cstring>

char *std::strerror(int errno**);**

Defined in

strerror.c and farstrerror.c in rts.src

Description

The strerror function returns the string “string error”. This function is supplied to provide ANSI C compatibility.

Far Version (C)

far char *far_strerror(int errno**);**

strftime — *Format Time*

Syntax for C

```
#include    <time.h>
```

size_t *strftime(char *s, size_t maxsize, const char *format,
const struct tm *timeptr**);**

Syntax for C++

```
#include    <ctime>
```

size_t *std::strftime(char *s, size_t maxsize, const char *format,
const struct tm *timeptr**);**

Defined in

strftime.c in rts.src

Description

The strftime function formats a time (pointed to by timeptr) according to a format string and returns the formatted time in the string s. Up to maxsize characters can be written to s. The format parameter is a string of characters that tells the strftime function how to format the time; the following list shows the valid characters and describes what each character expands to.

Character	Expands to
%a	The abbreviated *weekday* name (Mon, Tue, . . .)
%A	The full *weekday* name
%b	The abbreviated *month* name (Jan, Feb, . . .)
%B	The locale's full *month* name
%c	The *date* and *time* representation
%d	The *day* of the month as a decimal number (0–31)
%H	The *hour* (24-hour clock) as a decimal number (00–23)
%I	The *hour* (12-hour clock) as a decimal number (01–12)
%j	The *day* of the year as a decimal number (001–366)
%m	The *month* as a decimal number (01–12)
%M	The *minute* as a decimal number (00–59)
%p	The locale's equivalency of either *a.m.* or *p.m.*
%S	The *seconds* as a decimal number (00–50)
%U	The *week* number of the year (Sunday is the first day of the week) as a decimal number (00–52)
%x	The *date* representation
%X	The *time* representation
%y	The *year* without century as a decimal number (00–99)
%Y	The *year* with century as a decimal number
%Z	The *time zone* name, or by no characters if no time zone exists

For more information about the functions and types that the time.h/ctime header declares and defines, see section 8.4.12, *Time Functions (time.h/ ctime)*, on page 8-25.

Far Version

None

strlen — *Find String Length*

Syntax for C

#include <string.h>

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

Syntax for C++

#include <cstring>

size_t **std::strlen(**const char *string**);**

Defined in

strlen.c and far strlen.c in rts.src

Description

The strlen function returns the length of string. In C/C++, a character string is terminated by the first byte with a value of 0 (a null character). The returned result does not include the terminating null character.

Example

```
char *stra = ”who is there?”;
char *strb = ”abcdefghijklmnopqrstuvwxyz”;
char *strc = ”abcdefg”;
size_t length;

length = strlen(stra);      /* length = 13 */
length = strlen(strb);      /* length = 26 */
length = strlen(strc);      /* length = 7  */
```

Far Version (C)

size_t far_strlen(const far char *string**);**

strncat — *Concatenate Strings*

Syntax for C

#include <string.h>

char *strncat(char *dest, const char *src**,** size_t n**);**

Syntax for C++

#include <cstring>

char *std::strncat(char *dest, const char *src**,** size_t n**);**

Defined in

strncat.c and farstrncat.c in rts.src

Description

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

Example

In the following example, the character strings pointed to by *a, *b, and *c were assigned the values shown in the comments. In the comments, the notation \0 represents the null character:

```
char *a, *b, *c;
size_t size = 13;
.
.
.
/* a––> ”I do not like them,\0”             */;
/* b––> ” Sam I am, \0”                     */;
/* c––> ”I do not like green eggs and ham\0” */;
```

`strncat (a,b,size);`

```
/* a––> ”I do not like them, Sam I am, \0"   */;
/* b––> ” Sam I am, \0”                     */;
/* c––> ”I do not like green eggs and ham\0” */;
```

`strncat (a,c,size);`

```
/* a––> ”I do not like them, Sam I am, I do not like\0”   */;
/* b––> ” Sam I am, \0”                                  */;
/* c––> ”I do not like green eggs and ham\0”             */;
```

Far Version (C)

far char *far_strncat(far char *dest, const far char *src, size_t n**);**

strncmp — *Compare Strings*

Syntax for C

#include <string.h>

int strncmp(const char *string1, const char *string2, size_t n**);**

Syntax for C++

#include <cstring>

int std::strncmp(const char *string1, const char *string2, size_t n**);**

Defined in

strncmp.c and farstrncmp.c in rts.src

Description

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

< 0	if *string1 is less than *string2
0	if *string1 is equal to *string2
> 0	if *string1 is greater than *string2

Example

```
char *stra = ”why ask why”;
char *strb = ”just do it”;
char *strc = ”why not?”;
size_t size = 4;

if (strncmp(stra, strb, size) > 0)
   {
      /* statements here will get executed */
   }
if (strncmp(stra, strc, size) == 0)
   {
      /* statements here will get executed also */
   }
```

Far Version (C)

int far_strncmp(const far char *string1, const far char *string2, size_t n**);**

strncpy — *String Copy*

Syntax for C

#include <string.h>

char *strncpy(register char *dest, register const char *src**,**
register size_t n**);**

Syntax for C++

#include <cstring>

char *std::strncpy(register char *dest, register const char *src**,**
register size_t n**);**

Defined in

strncpy.c and farstrncpy.c in rts.src

Description

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

Example

Note that strb contains a leading space to make it five characters long. Also note that the first five characters of strc are an I, a space, the word am, and another space, so that after the second execution of strncpy, stra begins with the phrase I am followed by two spaces. In the comments, the notation \0 represents the null character.

```
char *stra = ″she′s the one mother warned you of″;
char *strb = ″ he′s″;
char *strc = ″I am the one father warned you of″;
char *strd = ″oops″;
int length = 5;
```

`strncpy (stra,strb,length);`

```
/* stra––> ″ he′s the one mother warned you of\0″ */;
/* strb––> ″ he′s\0″                              */;
/* strc––> ″I am the one father warned you of\0″  */;
/* strd––> ″oops\0″                               */;
```

`strncpy (stra,strc,length);`

```
/* stra––> ″I am  the one mother warned you of\0″ */;
/* strb––> ″ he′s\0″                              */;
/* strc––> ″I am the one father warned you of\0″  */;
/* strd––> ″oops\0″                               */;
```

`strncpy (stra,strd,length);`

```
/* stra––> ″oops\0″                               */;
/* strb––> ″ he′s\0″                              */;
/* strc––> ″I am the one father warned you of\0″  */;
/* strd––> ″oops\0″                               */;
```

Far Version (C)

far char *far_strncpy(far char *dest, const far char *src, size_t n**);**

strpbrk — *Find Any Matching Character*

Syntax for C

#include <string.h>

char *strpbrk(const char *string, const char *chs**);**

Syntax for C++

#include <cstring>

char *std::strpbrk(const char *string, const char *chs**);**

Defined in

strpbrk.c and farstrpbrk.c in rts.src

Description

The strpbrk function locates the first occurrence in string of *any* character in chs. If strpbrk finds a matching character, it returns a pointer to that character; otherwise, it returns a null pointer (0).

Example

```
char *stra = ”it wasn’t me”;
char *strb = ”wave”;
char *a;
```

```
a = strpbrk (stra,strb);
```

After this example, *a points to the w in wasn’t.

Far Version (C)

far char *far_strpbrk(const far char *string, const far char *chs**);**

strrchr — *Find Last Occurrence of a Character*

Syntax for C

#include <string.h>

char *strrchr(const char *string**,** int c**);**

Syntax for C++

#include <cstring>

char *std::strrchr(const char *string**,** int c**);**

Defined in

strrchr.c and farstrrchr.c in rts.src

Description

The strrchr function finds the last occurrence of c in string. If strrchr finds the character, it returns a pointer to the character; otherwise, it returns a null pointer (0).

Example

```
char *a = ”When zz comes home, the search is on for zs”;
char *b;
char the_z = ’z’;
```

After this example, *b points to the z in zs near the end of the string.

Far Version (C)

far char *far_strrchr(const far char *string, int c**);**

strspn — *Find Number of Matching Characters*

Syntax for C

#include <string.h>

size_t strspn(const char *string, const char *chs**);**

Syntax for C++

#include <cstring>

size_t std::strspn(const char *string, const char *chs**);**

Defined in

strspn.c and farstrspn.c in rts.src

Description

The strspn function returns the length of the initial segment of string, which is entirely made up of characters in chs. If the first character of string is not in chs, the strspn function returns 0.

Example

```
char *stra = ″who is there?″;
char *strb = ″abcdefghijklmnopqrstuvwxyz″;
char *strc = ″abcdefg″;
size_t length;

length = strcspn(stra,strb);    /* length = 3 */
length = strcspn(stra,strc);    /* length = 0 */
```

Far Version (C)

size_t far_strspn(const far char *string, const far char *chs**);**

strstr — *Find Matching String*

Syntax for C

#include <string.h>

char *strstr(const char *string1, const char *string2**);**

Syntax for C++

#include <cstring>

char *std::strstr(const char *string1, const char *string2**);**

Defined in

strstr.c and farstrstr.c in rts.src

Description

The strstr function finds the first occurrence of string2 in string1 (excluding the terminating null character). If strstr finds the matching string, it returns a pointer to the located string; if it does not find the string, it returns a null pointer. If string2 points to a string with length 0, strstr returns string1.

Example

```
char *stra = ″so what do you want for nothing?″;
char *strb = ″what″;
char *ptr;

ptr = strstr(stra,strb);
```

The pointer *ptr now points to the w in what in the first string.

Far Version (C)

far char *far_strstr(const far char *string1, const far char *string2**);**

strtod/strtol/ strtoul — *String to Number*

Syntax for C

#include <stdlib.h>

double **strtod(**const char *st**,** char **endptr**);**
long **strtol(**const char *st**,** char **endptr, int base**);**
unsigned long **strtoul(**const char *st, char **endptr, int base**);**

Syntax for C++

#include <cstdlib>

double **std::strtod(**const char *st**,** char **endptr**);**
long **std::strtol(**const char *st**,** char **endptr, int base**);**
unsigned long **std::strtoul(**const char *st, char **endptr, int base**);**

Defined in

strtod.c, strtol.c, and strtoul.c in rts.src
farstrtod.c, farstrtol.c, and farstrtoul.c in rts.src

Description

Three functions convert ASCII strings to numeric values. For each function, argument st points to the original string. Argument endptr points to a pointer; the functions set this pointer to point to the first character after the converted string.The functions that convert to integers also have a third argument, base, which tells the function the base in which to interpret the string.

- The strtod function converts a string to a floating-point value. The string must have the following format:

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

  The function returns the converted string; if the original string is empty or does not have the correct format, the function returns a 0. If the converted string would cause an overflow, the function returns ±HUGE_VAL; if the converted string would cause an underflow, the function returns 0. If the converted string overflows or underflows, errno is set to the value of ERANGE.

- The strtol function converts a string to a long integer. The string must have the following format:

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

- The strtoul function converts a string to an unsigned long integer. The string must have the following format:

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

The space is indicated by a horizontal or vertical tab, space bar, carriage return, form feed, or new line. Following the space is an optional sign and digits that represent the integer portion of the number. The fractional part of the number follows, then the exponent, including an optional sign.

The first unrecognized character terminates the string. The pointer that endptr points to is set to point to this character.

Far Version (C)

double far_strtod (const far char *st, far char **endptr**)**;
long far_strtol (const far char *st, far char **endptr, int base**)**;
unsigned long far_strtoul (const far char *st, far char **endprt, int base**);**

strtok

Break String into Token

Syntax for C

#include <string.h>

char *strtok(char *str1, const char *str2**);**

Syntax for C++

#include <cstring>

char *std::strtok(char *str1, const char *str2**);**

Defined in

strtok.c and farstrtok.c in rts.src

Description

Successive calls to the strtok function break str1 into a series of tokens, each delimited by a character from str2. Each call returns a pointer to the next token.

Example

After the first invocation of strtok in the following example, the pointer stra points to the string excuse\0 because strtok has inserted a null character where the first space used to be. In the comments, the notation \0 represents the null character.

```
char *stra = ″excuse me while I kiss the sky″;
char *ptr;

ptr = strtok (stra,″ ″); /* ptr −−> ″excuse\0″ */
ptr = strtok (0,″ ″);    /* ptr −−> ″me\0″     */
ptr = strtok (0,″ ″);    /* ptr −−> ″while\0″  */
```

Far Version (C)

far char *far_strtok(far char *str1, const far char *str2**);**

strxfrm — *Convert Characters*

Syntax for C

#include <string.h>

size_t strxfrm(char *to, const char *from, size_t n**);**

Syntax for C++

#include <cstring>

size_t std::strxfrm(char *to, const char *from, size_t n**);**

Defined in

strxfrm.c and farstrxfrm.c in rts.src

Description

The strxfrm function converts n characters pointed to by from into the n characters pointed to by to.

Far Version (C)

size_t far_strxfrm(far char *to, const far char *from, size_t n**);**

tan — *Tangent*

Syntax for C

#include <math.h>

double tan(double x**);**

Syntax for C++

#include <cmath>

double std::tan(double x**);**

Defined in

tan.c in rts.src

Description

The tan function returns the tangent of a floating-point number x. The angle x is expressed in radians. An argument with a large magnitude can produce a result with little or no significance.

Example

```
double x, y;

x = 3.1415927/4.0;
y = tan(x);           /* return value = 1.0 */
```

Far Version

None

tanh — *Hyperbolic Tangent*

Syntax for C

#include <math.h>

double tanh(double x**);**

Syntax for C++

#include <cmath>

double std::tanh(double x**);**

Defined in

tanh.c in rts.src

Description

The tanh function returns the hyperbolic tangent of a floating-point number x.

Example

```
double x, y;

x = 0.0;
y = tanh(x);        /* return value = 0.0 */
```

Far Version

None

time — *Time*

Syntax for C

#include <time.h>

time_t time(time_t *timer**);**

Syntax for C++

#include <ctime>

time_t std::time(time_t *timer**);**

Defined in

time.c in rts.src

Description

The time function determines the current calendar time, represented in seconds. If the calendar time is not available, the function returns –1. If timer is not a null pointer, the function also assigns the return value to the object that timer points to.

For more information about the functions and types that the time.h/ctime header declares and defines, see section 8.4.12, *Time Functions (time.h/ctime)*, on page 8-25.

Note: The time Function Is Target-System Specific

The time function is target-system specific, so you must write your own time function.

Far Version

None

tmpfile *Create Temporary File*

Syntax for C #include <stdlib.h>

FILE ***tmpfile(**void**);**

Syntax for C++ #include <cstdlib>

FILE ***std::tmpfile(**void**);**

Defined in tmpfile.c in rts.src

Description The tmpfile function creates a temporary file.

Far Version None

tmpnam *Generate Valid Filename*

Syntax for C #include <stdlib.h>

char *tmpnam(char *_s**);**

Syntax for C++ #include <cstdlib>

char *std::tmpnam(char *_s**);**

Defined in tmpnam.c in rts.src

Description The tmpnam function generates a string that is a valid filename.

Far Version None

toascii *Convert to ASCII*

Syntax for C #include <ctype.h>

int toascii(int c**);**

Syntax for C++ #include <cctype>

int std::toascii(int c**);**

Defined in toascii.c in rts.src

Description The toascii function ensures that c is a valid ASCII character by masking the lower seven bits. There is also an equivalent macro call _toascii.

Far Version None

tolower/toupper — *Convert Case*

Syntax for C

#include <ctype.h>

int tolower(int c**);**
int toupper(int c**);**

Syntax for C++

#include <cctype>

int std::tolower(int c**);**
int std::toupper(int c**);**

Defined in

tolower.c and toupper.c in rts.src

Description

Two functions convert the case of a single alphabetic character c into uppercase or lowercase:

❑ The tolower function converts an uppercase argument to lowercase. If c is already in lowercase, tolower returns it unchanged.

❑ The toupper function converts a lowercase argument to uppercase. If c is already in uppercase, toupper returns it unchanged.

The functions have macro equivalents named _tolower and _toupper.

Far Version

None

ungetc — *Write Character to Stream*

Syntax for C

#include <stdlib.h>

int ungetc(int c, FILE *_fp**);**

Syntax for C++

#include <cstdlib>

int std::ungetc(int c, FILE *_fp**);**

Defined in

ungetc.c in rts.src

Description

The ungetc function writes the character c to the stream pointed to by _fp.

Far Version

None

va_arg/va_end/ va_start

Variable-Argument Macros

Syntax for C

#include <stdarg.h>

typedef char ***va_list;**
va_arg(_ap**,** _type**);**
void va_end(_ap**);**
void va_start(_ap**,** parmN**);**

Syntax for C++

#include <cstdarg>

typedef char ***va_list;**
va_arg(_ap**,** _type**);**
void va_end(_ap**);**
void va_start(_ap**,** parmN**);**

Defined in

stdarg.h and cstdarg in rts.src

Description

Some functions are called with a varying number of arguments that have varying types. Such a function, called a variable-argument function, can use the following macros to step through its argument list at runtime. The _ap parameter points to an argument in the variable-argument list.

- The va_start macro initializes _ap to point to the first argument in an argument list for the variable-argument function. The parmN parameter points to the rightmost parameter in the fixed, declared list.
- The va_arg macro returns the value of the next argument in a call to a variable-argument function. Each time you call va_arg, it modifies _ap so that successive arguments for the variable-argument function can be returned by successive calls to va_arg (va_arg modifies _ap to point to the next argument in the list). The type parameter is a type name; it is the type of the current argument in the list.
- The va_end macro resets the stack environment after va_start and va_arg are used.

You must call va_start to initialize _ap before calling va_arg or va_end.

Example

```
int    printf (char *fmt...)
       va_list ap;
       va_start(ap, fmt);
       .
       .
       .
       i = va_arg(ap, int);     /* Get next arg, an integer */
       s = va_arg(ap, char *);  /* Get next arg, a string   */
       l = va_arg(ap, long);    /* Get next arg, a long     */
       .
       .
       .
       va_end(ap);              /* Reset                    */
}
```

Far Version None

vfprintf *Write to Stream*

Syntax for C #include <stdlib.h>

int vfprintf(FILE *_fp, const char *_format, va list _ap**);**

Syntax for C++ #include <cstdlib>

int std::vfprintf(FILE *_fp, const char *_format, va list _ap**);**

Defined in vfprintf.c in rts.src

Description The vfprintf function writes to the stream pointed to by _fp. The string pointed to by format describes how to write the stream. The argument list is given by _ap.

Far Version None

vprintf *Write to Standard Output*

Syntax for C #include <stdlib.h>

int vprintf(const char *_format, va list _ap**);**

Syntax for C++ #include <cstdlib>

int std::vprintf(const char *_format, va list _ap**);**

Defined in vprintf.c in rts.src

Description The vprintf function writes to the standard output device. The string pointed to by _format describes how to write the stream. The argument list is given by _ap.

Far Version None

vsprintf

Write Stream

Syntax for C

#include <stdlib.h>

int vsprintf(char *string, const char *_format, va list _ap**);**

Syntax for C++

#include <cstdlib>

int std::vsprintf(char *string, const char *_format, va list _ap**);**

Defined in

vsprintf.c in rts.src

Description

The vsprintf function writes to the array pointed to by _string. The string pointed to by _format describes how to write the stream. The argument list is given by _ap.

Far Version

None

Chapter 9

Library-Build Utility

When using the C/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.

The following prebuilt run-time-support libraries are included with the package:

- rts2800.lib (C/C++ runtime object library)
- rts2800_ml.lib (C/C++ large memory model run-time object library)

You can also build your own run-time-support libraries by using the mk2000 –v28 utility described in this chapter and the archiver described in the *TMS320C28x Assembly Language Tools User's Guide*.

Topic **Page**

9.1 Invoking the Library-Build Utility

The syntax for invoking the library-build utility is:

> **mk2000 –v28** [*options*] *src_arch1* [**–l***obj.lib1*] [*src_arch2* [**–l***obj.lib2*]] ...

mk2000 –v28	Command that invokes the utility.
options	Options affect how the library-build utility treats your files. Options can appear anywhere on the command line or in a linker command file. (Options are discussed in section 9.2 and section 9.3.)
src_arch	The name of a source archive file. For each source archive named, mk2000 –v28 builds an object library according to the runtime model specified by the command-line options.
–l*obj.lib*	The optional object library name. If you do not specify a name for the library, mk2000 –v28 uses the name of the source archive and appends a *.lib* suffix. For each source archive file specified, a corresponding object library file is created. You cannot build an object library from multiple source archive files.

The mk2000 –v28 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.

9.2 Library-Build Utility Options

Most of the options that are included on the command line correspond directly to options of the same name used with the compiler, assembler, linker, and shell. The following options apply only to the library-build utility.

––c	Extracts C source files contained in the source archive from the library and leaves them in the current directory after the utility completes execution.
––h	Uses header files contained in the source archive and leaves them in the current directory after the utility completes execution. Use this option to install the run-time-support header files from the rtsc.src or rtscpp.src archive that is shipped with the tools.
––k	Overwrites files. By default, the utility aborts any time it attempts to create an object file when an object file of the same name already exists in the current directory, regardless of whether you specified the name or the utility derived it.
––q	Suppresses header information (quiet).
––u	Uses existing header files (rather than 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 gives you flexibility in modifying run-time-support functions to suit your application.
––v	Prints progress information to the screen during execution of the utility. Normally, the utility operates silently (no screen messages).

For an online summary of the options, enter **mk2000 –v28** with no parameters on the command line.

9.3 Options Summary

The other options you can use with the library-build utility correspond directly to the options used with the compiler and assembler. Table 9–1 lists these options, and they are described in detail on the page indicated in the table.

Table 9–1. Summary of Options and Their Effects

(a) Options that control the compiler/shell

Option	**Effect**	**Page**
–d*name* [**=***def*]	Predefines *name*	2-13
–g	Enables symbolic debugging	2-13
–u*name*	Undefines *name*	2-14

(b) Options that control the parser

Option	**Effect**	**Page**
–pi	Disables definition-controlled inlining (but –o3 optimizations still perform automatic inlining)	2-37
–pk	Makes code K&R compatible	6-32
–pr	Enables relaxed mode; ignores strict ANSI violations	6-35
–ps	Enables strict ANSI mode (for C/C++, not K&R C)	6-35

(c) Options that control the diagnostics

Option	**Effect**	**Page**
–pdr	Issues remarks (nonserious warnings)	2-29
–pdv	Provides verbose diagnostics that display the original source with line wrap	2-30
–pdw	Suppresses warning diagnostics (errors are still issued)	2-30

(d) Options that control the optimization level

Option	**Effect**	**Page**
–o0	Compile with optimization register optimization	3-2
–o1	Compile with optimization + local optimization	3-3
–o2 (or –o)	Compile with optimization + global optimization	3-3
–o3	Compile with optimization + file optimization Note that mk2000 –v28 automatically sets –ol0 and –op0.	3-3

Table 9–1. Options Summary (Continued)

(e) Options that are machine specific

Option	Effect	Page
–ma	Assumes aliased variables	3-11
–md	Disables DP load optimization	2-14
–me	Disables generation of fast branch instructions	2-14
–mf	Optimizes for speed instead of for space	2-14
–mi	Disables generation of RPT instructions	2-14
–ml	Generates large memory model code	6-18
–mn	Enables optimization disabled by –g	2-15
–ms	Optimizes for size over speed	2-15, 3-17
–mt	Enables unified memory	2-15
–v28	Generates TMS320C28x architecture code	2-15

(f) Option that controls the assembler

Option	Effect	Page
–as	Keeps labels as symbols	2-19

(g) Options that change the default file extensions

Option	Effect	Page
–ea[.]*extension*	Sets default extension for assembly files	2-17
–eo[.]*extension*	Sets default extension for object files	2-17

Chapter 10

C++ Name Demangler Utility

The C++ compiler implements function overloading, operator overloading, and type-safe linking by encoding a function's signature in its link-level name. The process of encoding the signature into the linkname is often referred to as *name mangling*. When you inspect mangled names, such as in assembly files or linker output, it can be difficult to associate a mangled name with its corresponding name in the C++ source code. The C++ *name demangler* is a debugging aid that translates each mangled name it detects to its original name found in the C++ source code.

This chapter tells you how to invoke and use the C++ name demangler. The C++ name demangler reads in input, looking for mangled names. All unmangled text is copied to output unaltered. All mangled names are demangled before being copied to output.

Topic **Page**

10.1 Invoking the C++ Name Demangler

To invoke the C++ name demangler, enter:

```
dem2000 [options] [filenames]
```

dem2000	Command that runs the name demangler
options	Options that affect the way the demangler behaves. (Options are discussed in section 10.2, *C++ Name Demangler Options*.)
filenames	Text input files, such as the assembly file output by the compiler, the assembler listing file, and the linker map file. If no filenames are specified on the command line, dem2000 uses standard in.

By default, the C++ name demangler outputs to standard out. You can use the –o *file* option if you want to output to a file.

10.2 C++ Name Demangler Options

Following are the options that control the C++ name demangler, along with descriptions of their effects.

–h	Prints a help screen that provides an online summary of the C++ name demangler options
–o *file*	Outputs to the given *file* rather than to standard out
–u	External names do not have a C++ prefix
–v	Enables verbose mode (output banner)

10.3 Sample Usage of the C++ Name Demangler

Example 10–1 (a) shows a sample C++ program and the resulting assembly that is output by the TMS320C28x compiler. In Example 10–1 (b), the linknames of foo( ) and compute( ) are mangled; that is, their signature information is encoded into their names.

Example 10–1. Name Mangling

(a) C++ program

```
int compute(int val, int *err);

int foo(int val, int *err)
{
   static int last_err = 0;
   int        result   = 0

   if (last_err == 0)
   result = compute(val, &last_err);

   *err = last_err;
   return result;
}
```

Example 10–1. Name Mangling (Continued)

(b) Resulting assembly for foo()

```
;***************************************************************
;* FNAME: _foo_FiPi                     FR SIZE:   4           *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                            0 Parameter,  3 Auto,  0 SOE     *
;***************************************************************

_foo_FiPi:
       ADDB       SP,#4
       MOVZ       DP,#_last_err$1
       MOV        *–SP[1],AL            ; |4|
       MOV        AL,@_last_err$1       ; |8|
       MOV        *–SP[2],AR4           ; |4|
       MOV        *–SP[3],#0            ; |6|
       BF         L1,NEQ                ; |8|
       ; branch occurs ; |8|
       MOVL       XAR4,#_last_err$1     ; |9|
       MOV        AL,*–SP[1]            ; |9|
       LCR        #_compute__FiPi       ; |9|
       ; call occurs [#_compute__FiPi]  ; |9|
       MOV        *–SP[3],AL            ; |9|
L1:
       MOVZ       AR6,*–SP[2]           ; |11|
       MOV        *+XAR6[0],*(0:_last_err$1) ; |11|
       MOV        AL,*–SP[3]            ; |12|
       SUBB       SP,#4                 ; |12|
       LRETR
       ; return occurs
```

Executing the C++ name demangler utility demangles all names that it believes to be mangled. If you enter:

% dem2000 foo.asm

the result is shown in Example 2–2. Notice that the linknames of foo( ) and compute( ) are demangled.

Example 10–2. Result After Running the C++ Name Demangler Utility

```
;***************************************************************
;* FNAME: foo(int, int *)               FR SIZE:   4           *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                             0 Parameter,  3 Auto,  0 SOE     *
;***************************************************************

foo(int, int *):
       ADDB       SP,#4
       MOVZ       DP,#_last_err$1
       MOV        *–SP[1],AL            ; |4|
       MOV        AL,@_last_err$1      ; |8|
       MOV        *–SP[2],AR4           ; |4|
       MOV        *–SP[3],#0            ; |6|
       BF         L1,NEQ                ; |8|
       ; branch occurs ; |8|
       MOVL       XAR4,#_last_err$1    ; |9|
       MOV        AL,*–SP[1]            ; |9|
       LCR        #compute(int, int *)       ; |9|
       ; call occurs [#compute(int, int *)]  ; |9|
       MOV        *–SP[3],AL            ; |9|
L1:
       MOVZ       AR6,*–SP[2]           ; |11|
       MOV        *+XAR6[0],*(0:_last_err$1) ; |11|
       MOV        AL,*–SP[3]            ; |12|
       SUBB       SP,#4                 ; |12|
       LRETR
       ; return occurs
```

Appendix A

Invoking the Compiler Tools Individually

The TMS320C28x™ C/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**

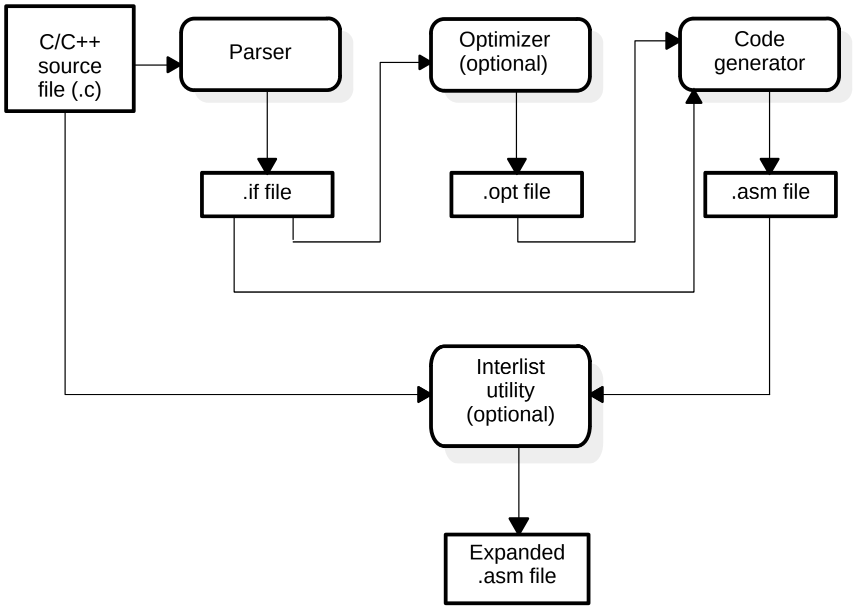
A.1 Which Tools Can be Invoked Individually

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/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/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/C++ source file. The utility produces an expanded assembly source file containing statements from the C/C++ file as assembly language comments. Section 2.10, *Using the Interlist Utility*, on page 2-39 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 *TMS320C28x 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/C++ Code,* describes how to run the linker. The linker is described fully in the *TMS320C28x Assembly Language Tool User's Guide.*

A.2 Invoking the Parser Individually

The first step in compiling a TMS320C28x C/C++ program is to invoke the 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:

ac2000 *input file* [*output file*] [*options*]

ac2000	is the command that invokes the parser.
input file	names the C/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

		See	
Parser Option	**Function**	**Shell Option**	**Page**
–Dd*name* [=def]	Predefines macro *name*	–d*name* [=def]	2-13
–I*dir* (capital i)	Defines #include search path	–i*dir*	2-13, 2-24
–K	Allows K&R compatibility	–pk	6-32
–C	Preprocess only, keeps comments and generates .pp file	–ppc	2-25
–M	Preprocess only, writes dependency lines	–ppd	2-25
–H	Preprocess only, writes list of #include files	–ppi	2-25
–P	Preprocess only, no comments	–ppo	2-10
–L	Preprocess only, writes line control information	–ppl	2-10
–q	Suppresses progress messages (quiet)	–q	2-13
–U*name*	Undefines macro *name*	–u*name*	2-14
–w	Suppresses warning messages	–pdw	2-10

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 –ppo option, specifying preprocessing only.

   ```
   cl2000 –v28 –ppo file.c
   ```

   If you want to use the –d, –u, or –i options, use them on this first pass. This pass produces a preprocessed output file called file.pp. For more information about the preprocessor, see section 2.5, *Controlling the Preprocessor*, on page 2-23.

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

   ```
   cl2000 –v28 file.pp
   ```

   You can use any other options on this final pass.

A.4 Invoking the Optimizer Individually

The second step in compiling a TMS320C28x C/C++ program—optimizing—is optional. After parsing a C/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/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:

opt2000 [*input file* [*output file*]] [*options*]

opt2000	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	6-32
–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, 2-39

A.5 Invoking the Code Generator Individually

The third step in compiling a TMS320C28x C/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:

cg2000 [*input file* [*output file* [*tempfile*]]] [*options*]

cg2000	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[†]	2-13
–q	Suppresses progress messages (quiet)	–q	2-13
–s	Optimizes for space instead of for speed	–mf	2-8
–v28	Generates TMS320C28x architecture code	–v28	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 TMS320C28x C/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/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:

```
cl2000 –v28 –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 TMS320C28x byte is 16 bits.

Note: TMS320C28x Byte Is 16 Bits

By ANSI C definition, the size of 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 TMS320C28x 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). TMS320C28x 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 TMS320C28x 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 TMS320C28x, 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.

load time 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.

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 target 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. TMS320C28x 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

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

run-time 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.

run-time 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 TMS320C28x-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