TMS320C55x Optimizing C/C++ Compiler User's Guide

Literature Number: SPRU281C
June 2001

Printed on Recycled Paper

IMPORTANT NOTICE

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

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

Customers are responsible for their applications using TI components.

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

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

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that products or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products.
www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Preface

Read This First

About This Manual

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

- Compiler
- Optimizer
- Interlist utility
- Library-build utility
- C++ name demangler

The TMS320C55x™ C/C++ compiler accepts C++ as well as American National Standards Institute (ANSI) standard C source code and produces assembly language source code for the TMS320C55x device.

This user's guide discusses the characteristics of the C/C++ compiler. It assumes that you already know how to write C programs. *The C Programming Language* (second edition), by Brian W. Kernighan and Dennis M. Ritchie, describes C based on the ANSI C standard. 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 the first edition of Kernighan and Ritchie's *The C Programming Language*.

Before you use the information about the C/C++ compiler in this user's guide, you should install the C/C++ compiler tools.

Notational Conventions

This document uses the following conventions:

- The TMS320C55x device is referred to as C55x.
- 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:

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

- 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 face 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:

  cl55 [*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:

  cl55 [*options*] [*filenames*] [*–z* [*link_options*] [*object files*]]

  The **cl55** command has several optional parameters.

- 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 that are not included in the actual syntax but indicate that you must specify either the –c or –cr option:

  lnk55 {**–c** | **–cr**} *filenames* [–o *name.out*] **–l** *libraryname*

Related Documentation From Texas Instruments

The following books describe the TMS320C55x and related support tools.

TMS320C55x Assembly Language Tools User's Guide (literature number SPRU280) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for TMS320C55x™ devices.

TMS320C55x DSP CPU Reference Guide (literature number SPRU371) describes the architecture, registers, and operation of the CPU for the TMS320C55x™ digital signal processors (DSPs). This book also describes how to make individual portions of the DSP inactive to save power.

TMS320C55x DSP Mnemonic Instruction Set Reference Guide (literature number SPRU374) describes the TMS320C55x™ DSP mnemonic instructions individually. Also includes a summary of the instruction set, a list of the instruction opcodes, and a cross-reference to the algebraic instruction set.

TMS320C55x DSP Algebraic Instruction Set Reference Guide (literature number SPRU375) describes the TMS320C55x™ DSP algebraic instructions individually. Also includes a summary of the instruction set, a list of the instruction opcodes, and a cross-reference to the mnemonic instruction set.

TMS320C55x Programmer's Guide (literature number SPRU376) describes ways to optimize C and assembly code for the TMS320C55x™ DSPs and explains how to write code that uses special features and instructions of the DSP.

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:

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

ISO/IEC 14882-1998, the International Standard for Programming Language C++, American National Standards Institute

C: A Reference Manual (fourth edition), by Samuel P. Harbison, and Guy L. Steele Jr., published by Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

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

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

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

The C++ Programming Language (second edition), Bjarne Stroustrup, published by Addison-Wesley Publishing Company, Reading, Massachusetts, 1990

Trademarks

Contents

Figures

Tables

Examples

Chapter 1

Introduction

The TMS320C55x™ is supported by a set of software development tools, which includes an optimizing C/C++ compiler, an assembler, a linker, and assorted utilities.

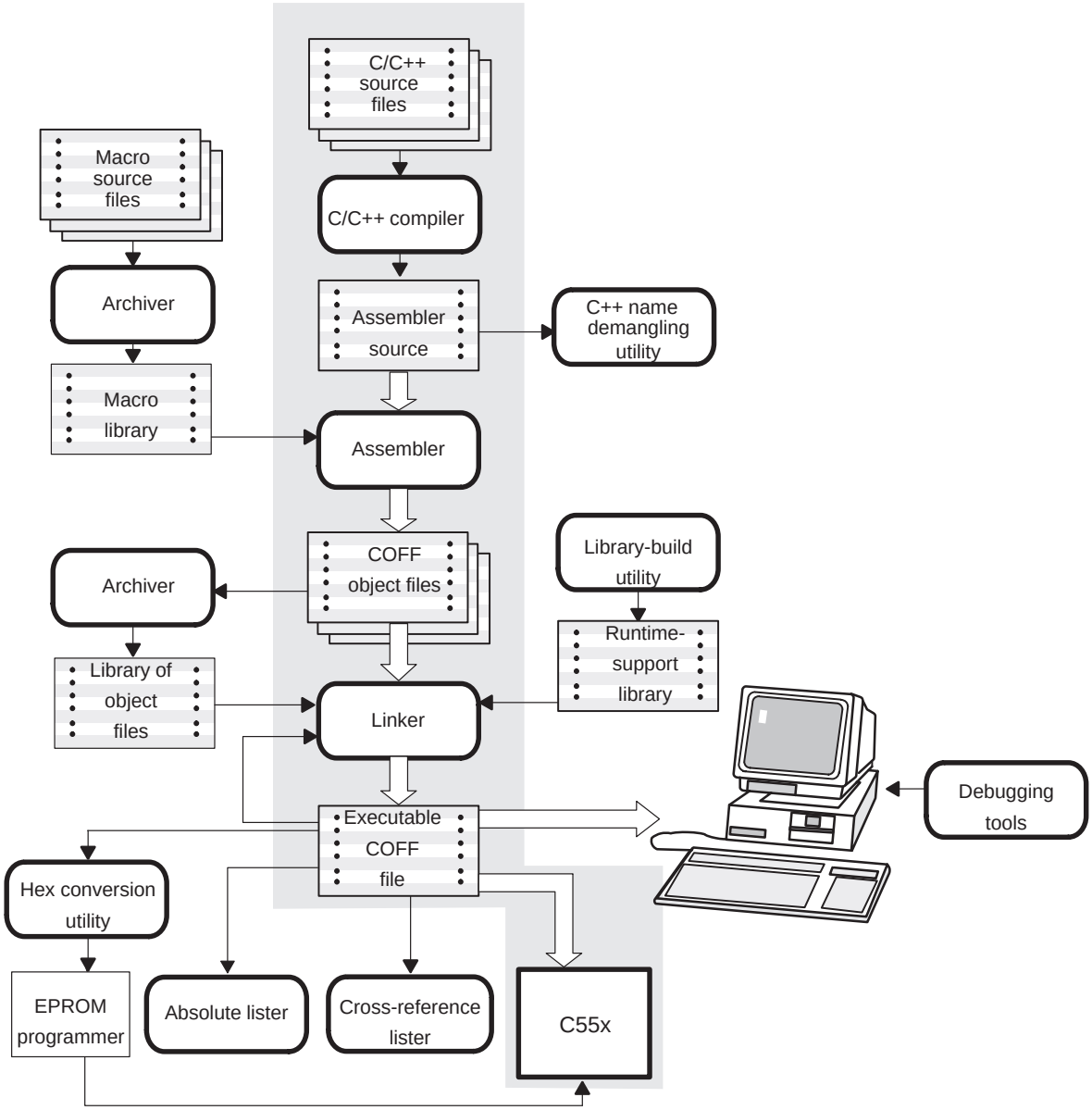
This chapter provides an overview of these tools and introduces the features of the optimizing C/C++ compiler. The assembler and linker are discussed in detail in the *TMS320C55x Assembly Language Tools User's Guide*.

Topic **Page**

1.1 Software Development Tools Overview

Figure 1–1 illustrates the C55x 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. TMS320C55x 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 C55x assembly language source code. A **shell program**, an **optimizer**, and an **interlist utility** are parts of the compiler:
  - The shell program enables you to compile, assemble, and link source modules in one step.
  - The optimizer modifies code to improve the efficiency of C/C++ programs.
  - The interlist utility interweaves C/C++ source statements with assembly language output.

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

- The **assembler** translates assembly language source files into machine language object files. The TMS320C55x tools include two assemblers. The mnemonic assembler accepts 'C54x and C55x mnemonic assembly source files. The algebraic assembler accepts C55x algebraic assembly source files. The machine language is based on common object file format (COFF). The TMS320C55x *Assembly Language Tools User's Guide* explains how to use the assembler.

- The **linker** combines object files into a single executable object module. As it creates the executable module, it performs relocation and resolves external references. The linker accepts relocatable COFF object files and object libraries as input. See Chapter 4, *Linking C Code*, for information about invoking the linker. See the *TMS320C55x Assembly Language Tools User's Guide* for a complete description of the linker.

- The **archiver** allows you to collect a group of files into a single archive file, called a *library*. Additionally, the archiver allows you to modify a library by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is building a library of object modules. The *TMS320C55x Assembly Language Tools User's Guide* explains how to use the archiver.

- You can use the **library-build utility** to build your own customized runtime-support library (see Chapter 8, *Library-Build Utility*). Standard runtime-support library functions are provided as source code in rts.src.

  The **runtime-support libraries** contain the ANSI standard runtime-support functions, compiler-utility functions, floating-point arithmetic functions, and C I/O functions that are supported by the C55x compiler. See Chapter 7, *Runtime-Support Functions*, for more information.

- The C55x debugger accepts executable COFF files as input, but most EPROM programmers do not. The **hex conversion utility** converts a COFF object file into TI-Tagged, ASCII-hex, Intel, Motorola-S, or Tektronix object format. The converted file can be downloaded to an EPROM programmer. The *TMS320C55x Assembly Language Tools User's Guide* explains how to use the hex conversion utility.

- The **absolute lister** accepts linked object files as input and creates .abs files as output. You can assemble these .abs files to produce a listing that contains absolute, rather than relative, addresses. Without the absolute lister, producing such a listing would be tedious and would require many manual operations. The *TMS320C55x Assembly Language Tools User's Guide* explains how to use the absolute lister.

- The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definitions, and their references in the linked source files. The *TMS320C55x Assembly Language Tools User's Guide* explains how to use the cross-reference lister.

- The main product of this development process is a module that can be executed in a TMS320C55x device. You can use one of several debugging tools to refine and correct your code. Available products include:

  - An instruction-accurate software simulator
  - An extended development system (XDS510™) emulator

  These tools are accessed within Code Composer Studio. For more information, see the *Code Composer Studio User's Guide*.

1.2 C/C++ Compiler Overview

The C55x C/C++ compiler is a full-featured optimizing compiler that translates standard ANSI C/C++ programs into C55x assembly language source. The following subsections describe the key features of the compiler.

1.2.1 ANSI Standard

The following features pertain to ANSI standards:

- **ANSI-standard C**

  The C55x C/C++ compiler fully conforms to the ANSI C standard as defined by the ANSI specification and described in the second edition of Kernighan and Ritchie's *The C Programming Language* (K&R). The ANSI C standard includes extensions to C that provide maximum portability and increased capability.

- **C++**

  The C55x C/C++ compiler also supports C++ as defined by Ellis and Stroustrup's *The Annotated C++ Reference Manual* (ARM) and many of the features in the ISO/IEC 14882–1998 standard for C++.

- **ANSI-standard runtime support**

  The compiler tools come with a complete runtime library. All library functions conform to the ANSI C library standard. The library includes functions for standard input and output, string manipulation, dynamic memory allocation, data conversion, time-keeping, trigonometry, and exponential and hyperbolic functions. Functions for signal handling are not included, because these are target-system specific.

  The C++ library includes the ANSI C subset as well as those components necessary for language support.

  For more information, see Chapter 7, *Runtime-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 easily, 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 supports 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 tools include 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, so you can write assembly and C functions that call each other. For more information, see Chapter 6, *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, allowing for faster compilation. Standalone preprocessing or preprocessed listing is also available. For more information, see Section 2.5, *Controlling the Preprocessor*, on page 2-25.

- **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, and C55x specific optimizations take advantage of the features specific to the C55x 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 tools include a utility that interlists your original C/C++ source statements into the assembly language output of the compiler. This utility provides you with a 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 2-42.

- **Library-build utility**

  The library-build utility lets you custom-build object libraries from source for any combination of runtime models or target CPUs. For more information, see Chapter 8, *Library-Build Utility*.

1.3 The Compiler and Code Composer Studio

Code Composer Studio provides a graphical interface for using the code generation tools.

A Code Composer Studio project keeps track of all information needed to build a target program or library. A project records:

- Filenames of source code and object libraries
- Compiler, assembler, and linker options
- Include file dependencies

When you build a project with Code Composer Studio, the appropriate code generation tools are invoked to compile, assemble, and/or link your program.

Compiler, assembler, and linker options can be specified within Code Composer Studio's Build Options dialog. Nearly all command line options are represented within this dialog. Options that are not represented can be specified by typing the option directly into the editable text box that appears at the top of the dialog.

The information in this book describes how to use the code generation tools from the command line interface. For information on using Code Composer Studio, see the *Code Composer Studio User's Guide*. For information on setting code generation tool options within Code Composer Studio, see the Code Generation Tools online help.

Chapter 2

Using the C/C++ Compiler

Translating your source program into code that the TMS320C55x™ can execute is a multi-step process. You must compile, assemble, and link your source files to create an executable object file. The C55x compiler tools contain a special shell program, cl55, 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 lets you compile, assemble, and optionally link in one step. The shell runs one or more source modules through the following:

- The **compiler**, which includes the parser, optimizer, and code generator, accepts C/C++ source code and produces C55x assembly language source code.

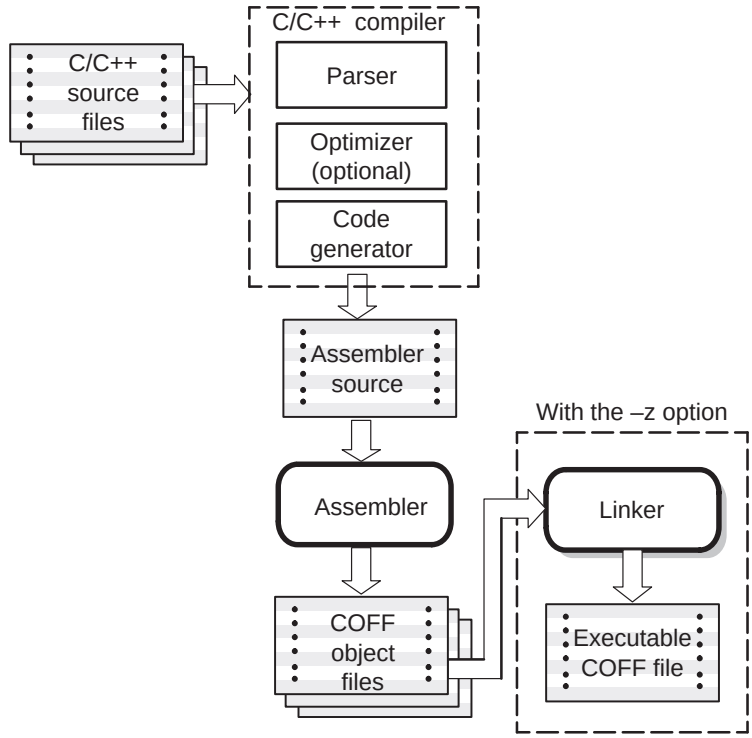
  You can compile C and C++ files in a single command—the compiler uses the conventions for filename extensions to distinguish between them (see section 2.3.3, *Specifying Filenames*, for more information).

- The **assembler** generates a COFF object file.

- 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 a complete description of the assembler and the linker, see the *TMS320C55x Assembly Language Tools User's Guide*.

2.2 Invoking the C/C++ Compiler Shell

To invoke the compiler shell, enter:

cl55 [*options*] [*filenames*] [**–z** [*link_options*] [*object files*]]

cl55	Command that runs the compiler and the assembler
options	Options that affect the way the shell processes input files (the options are listed in Table 2–1 on page 2-7)
filenames	One or more C/C++ source files, assembly source files, or object files.
–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
object files	Name of the additional object files for the linking process

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

```
cl55 –q symtab.c file.c seek.asm
```

As cl55 encounters each source file, it prints the C/C++ filenames in square brackets ([ ]) and assembly language filenames in angle brackets (< >). This example uses the –q option to suppress the additional progress information that cl55 produces. Entering the command above produces these messages:

```
[symtab.c]
[file.c]
<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:

```
% cl55 symtab.c
TMS320C55x ANSI C/C++ Compiler              Version x.xx
Copyright (c) 2001          Texas Instruments Incorporated
   ”symtab.c”   ==> main
TMS320C55x ANSI C/C++ Codegen               Version x.xx
Copyright (c) 2001          Texas Instruments Incorporated
   ”symtab.c”:  ==> main
TMS320C55x COFF Assembler                Version x.xx
Copyright (c) 2001          Texas Instruments Incorporated
 PASS 1
 PASS 2
```

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 sequences of letters.
- 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, –pi, –pk, and –pl can be combined as –pikl.
- Options that have parameters, such as –u*name* and –i*directory*, cannot be combined. They must be specified separately.
- An option with a required parameter can be specified with or without a space separating the parameter from the option. For example, the option to undefine a name can be specified as –u *name* or –u*name*. However, a required numeric parameter must be placed immediately after the option (no space between the option and parameter).
- An option with an optional parameter must be specified with the parameter immediately after the option (no space between the option and parameter). For example, the option to specify the maximum amount of optimization must be specified as –o3, not –o 3.
- Files and options can occur in any order except the –z option. The –z option must follow all other compiler options and precede any linker options.

You can define default options for the shell by using the C_OPTION or C55X_C_OPTION environment variable. For more information on the C_OPTION environment variable, see subsection 2.4.2, *Setting Default Shell Options (C_OPTION and C55X_C_OPTION)*, on page 2-23.

Table 2–1 summarizes all options (including linker options). Use the page references in the table for more complete descriptions of the options.

For an online summary of the options, enter cl55 with no parameters on the command line.

Table 2–1. Shell Options Summary

(a) Options that control the compiler shell

Option	Effect	Page(s)
–@*filename*	Interprets contents of a file as an extension to the command line	2-14
–b	Generates auxiliary information file	2-14
–c	Disables linking (negate –z)	2-14, 4-5
–d*name*[=*def*]	Predefines *name*	2-14
–g	Enables symbolic debugging	2-14
–gw	Enables symbolic debugging, using the DWARF debug format in the object file. Compile with this option if your application contains C++ source files.	2-14
–i*directory*	Defines #include search path	2-15, 2-26
–k	Keeps .asm file	2-15
–n	Compiles only	2-15
–q	Suppresses progress messages (quiet)	2-15
–qq	Suppresses all messages (super quiet)	2-15
–s	Interlists optimizer comments (if available) and assembly statements; otherwise interlists C/C++ source and assembly statements	2-16
–ss	Interlists C/C++ source and assembly statements	2-16, 3-13
–u*name*	Undefines *name*	2-16
–v*device*[:*revision*]	Causes the compiler to generate optimal code for the C55x device and optional revision number specified.	2-16
–z	Enables linking	2-16

Table 2–1. Shell Options Summary (Continued)

(b) Options that change the default file extensions when creating a file

Option	Effect	Page
–ea[.]*newextension*	Sets default extension for assembly files	2-19
–eo[.]*newextension*	Sets default extension for object files	2-19
–ec[.]*newextension*	Sets default extension for C source files	2-19
–ep[.]*newextension*	Sets default extension for C++ source files	2-19
–es[.]*newextension*	Sets default extension for assembly listing files	2-19

(c) Options that specify file and directory names

Option	Effect	Page
–fa*filename*	Identifies *filename* as an assembly source file, regardless of its extension. By default, the compiler treats .asm files as assembly source files.	2-18
–fc*filename*	Identifies *filename* as a C source file, regardless of its extension. By default, the compiler treats .c files as C source files.	2-18
–fg*filename*	Processes a C *filename* as a C++ file.	2-18
–fo*filename*	Identifies *filename* as an object code file, regardless of its extension. By default, the compiler and linker treat .obj files as object code files.	2-18
–fp*filename*	Identifies *filename* as a C++ file, regardless of its extension. By default, the compiler treats .C, .cpp, .cc, or .cxx files as C++ files.	2-18

(d) Options that specify directories

Option	Effect	Page
–fb*directory*	Specifies absolute listing file directory	2-20
–ff*directory*	Specifies an assembly listing and cross-reference listing file directory	2-20
–fr*directory*	Specifies object file directory	2-20
–fs*directory*	Specifies assembly file directory	2-20
–ft*directory*	Specifies temporary file directory	2-20

Table 2–1. Shell Options Summary (Continued)

(e) Options that control parsing

Option	Effect	Page
–pe	Enables embedded C++ mode	5-33
–pi	Disables definition-controlled inlining (but –o3 optimizations still perform automatic inlining)	2-38
–pk	Allows K&R compatibility	5-31
–pl	Generates a raw listing file	2-35
–pm	Combines source files to perform program-level optimization	3-6
–pr	Enables relaxed mode; ignores strict ANSI violations	5-33
–ps	Enables strict ANSI mode (for C/C++, not K&R C)	5-33
–px	Generates a cross-reference listing file	2-34
–rtti	Enables runtime type information (RTTI). RTTI allows the type of an object to be determined at runtime.	5-5

(f) Parser options that control preprocessing

Option	Effect	Page
–ppa	Continues compilation after preprocessing	2-27
–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	2-27
–ppd	Performs preprocessing only, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility	2-28
–ppi	Performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive	2-28
–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	2-27
–ppo	Performs preprocessing only. Writes preprocessed output to a file with the same name as the input but with a .pp extension	2-27

Table 2–1. Shell Options Summary (Continued)

(g) Parser 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.)	2-31
–pden	Displays a diagnostic's identifiers along with its text	2-31
–pdf	Generates a diagnostics information file	2-31
–pdr	Issues remarks (nonserious warnings)	2-31
–pds*num*	Suppresses the diagnostic identified by *num*	2-31
–pdse*num*	Categorizes the diagnostic identified by *num* as an error	2-31
–pdsr*num*	Categorizes the diagnostic identified by *num* as a remark	2-31
–pdsw*num*	Categorizes the diagnostic identified by *num* as a warning	2-31
–pdv	Provides verbose diagnostics that display the original source with line-wrap	2-32
–pdw	Suppresses warning diagnostics (errors are still issued)	2-32

(h) Options that are C55x-specific

Option	Effect	Page
–ma	Indicates that a specific aliasing technique is used	3-11
–mb	Specifies that all data memory will reside on-chip	2-15
–mc	Allows constants normally placed in a .const section to be treated as read-only, initialized static variables	2-15
–mg	Accepts C55x algebraic assembly files	2-15
–ml	Uses the large memory model	6-3
–mn	Enables optimizations disabled by –g	3-15
–mr	Prevents the compiler from generating hardware blockrepeat, localrepeat, and repeat instructions. Only useful when –o2 or –o3 is also specified.	2-15
–ms	Optimize for minimum code space	2-15

Table 2–1. Shell Options Summary (Continued)

(i) Options that control optimization

Option	Effect	Page
–o0	Optimizes register usage	3-2
–o1	Uses –o0 optimizations and optimizes locally	3-2
–o2 or –o	Uses –o1 optimizations and optimizes globally	3-3
–o3	Uses –o2 optimizations and optimizes file	3-3
–oi*size*	Sets automatic inlining size (–o3 only)	3-12
–ol0 (–oL0)	Informs the optimizer that your file alters a standard library function	3-4
–ol1 (–oL1)	Informs the optimizer that your file declares a standard library function	3-4
–ol2 (–oL2)	Informs the optimizer that your file does not declare or alter library functions. Overrides the –ol0 and –ol1 options	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	2-21
–ac	Makes case significant in assembly source files	2-21
–ad*name*	Sets the *name* symbol	2-21
–ahc *filename*	Copies the specified file for the assembly module	2-21
–ahi *filename*	Includes the file for the assembly module	2-21
–al	Generates an assembly listing file	2-21
–ar*num*	Suppresses the assembler remark identified by *num*	2-21
–as	Puts labels in the symbol table	2-21
–ata	Informs the assembler that the ARMS status bit will be enabled during the execution of this source file	2-21
–atc	Informs the assembler that the CPL status bit will be enabled during the execution of this source file	2-22
–ath	Causes the assembler to encode C54x instructions for speed over size	2-22
–atl	Informs the assembler that the C54CM status bit will be enabled during the execution of this source file	2-22
–atn	Causes the assembler to remove NOPs located in the delay slots of C54x delayed branch/call instructions	2-22
–att	Informs the assembler that the SST status bit will be disabled during the execution of this source file	2-22
–atv	Causes the assembler to use the largest form of certain variable-length instructions	2-22
–atw	Suppresses assembler warning messages (supported for the algebraic assembler only)	2-22
–au*name*	Undefines the predefined constant *name*	2-22
–ax	Generates the cross-reference file	2-22
—purecirc	Causes the assembler to encode for C54x-specific circular addressing	2-22

Table 2–1. Shell Options Summary (Continued)

(k) Options that control the linker

Options	Effect	Page
–a	Generates absolute output	4-6
–abs	Produces an absolute listing file. Use this option only after specifying the –z option.	2-14
–ar	Generates relocatable output	4-6
–b	Disables merge of symbolic debugging information	4-6
–c	Autoinitializes variables at runtime	4-6
–cr	Autoinitializes variables at reset	4-6
–e *global_symbol*	Defines entry point	4-6
–f *fill_value*	Defines fill value	4-6
–g *global_symbol*	Keeps a *global_symbol* global (overrides –h)	4-6
–h	Makes global symbols static	4-6
–heap *size*	Sets heap size (bytes)	4-6
–i *directory*	Defines library search path	4-6
–l *filename*	Supplies library name	4-6
–m *filename*	Names the map file	4-6
–o *filename*	Names the output file	4-7
–q	Suppresses progress messages (quiet)	4-7
–r	Generates relocatable output	4-7
–s	Strips symbol table	4-7
–stack *size*	Sets primary stack size (bytes)	4-7
–sysstack *size*	Sets secondary system stack size (bytes)	4-7
–u *symbol*	Undefines symbol	4-7
–w	Displays a message when an undefined output section is created	4-7
–x	Forces rereading of libraries	4-7

2.3.1 Frequently Used Options

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

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

Within the command file, filenames or option parameters containing embedded spaces or hyphens must be surrounded by quotation marks. For example: "this-file.obj"

–abs Produces an absolute listing file when used after the –z option. Note that you must use the –o option (after –z) to specify the .out file for the absolute lister, even if you use a linker command file that already uses –o.

–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 an .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 or C55X_C_OPTION environment variable and you don't want to link. For more information, see subsection 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 debuggers and enables assembly source debugging in the assembler.

–gw Generates DWARF symbolic debugging directives that are used by the C/C++ source-level debugger and enables assembly source debugging in the assembler. Use this option to generate debug information when your application contains C++ source files. For more information on the DWARF debug format, see the *DWARF Debugging Information Format Specification*, 1992–1993, UNIX International, Inc.

–i*directory* Adds *directory* to the list of directories that the compiler searches for #include files. You can use this option a maximum of 32 times to define several directories; be sure to separate –i options with spaces. If you don't specify a directory name, the preprocessor ignores the –i option. For more information, see subsection 2.5.2.1, *Changing the #include File Search Path With the –i Option*, on page 2-26.

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

–mb Specifies that all data memory will reside on-chip. This option allows the compiler to optimize using the dual MAC instruction. You must ensure that your linker command file places all such data into on-chip memory.

–mc Allows constants that are normally placed in a .const section to be treated as read-only, initialized static variables. This allows the constant values to be loaded into extended memory while the space used to hold the values at runtime is still in page 0.

–mg By default, the compiler and assembler use mnemonic assembly: the compiler generates mnemonic assembly output, and the assembler will only accept mnemonic assembly input files. When this option is specified, the compiler and assembler use algebraic assembly. You must use this option to assemble algebraic assembly input files, or to compile C code containing algebraic asm statements. Algebraic and mnemonic source code cannot be mixed in a single source file.

–mr Prevents the compiler from generating the hardware block-repeat, localrepeat, and repeat instructions. This option is only useful when –o2 or –o3 is specified.

–ms Optimizes for code space instead of for speed.

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

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

–qq Suppresses all output except error messages.

–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/C++ source statements are interlisted with the assembly language output of the compiler, which allows you to inspect the code generated for each C/C++ statement. When the optimizer is invoked (–o*n* option) along with this option, your code might be reorganized substantially. The –s option implies the –k option. For more information about using the interlist utility with the optimizer, see Section 3.7, *Using the Interlist Utility With the Optimizer*, on page 3-13.

–ss Invokes the interlist utility, which interweaves original C/C++ source with compiler-generated assembly language. If the optimizer is invoked (–o*n* option) along with this option, your code might be reorganized substantially. For more information, see Section 2.10, *Using the Interlist Utility,* on page 2-42.

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

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

2.3.2 Selecting the Device Version (–v Option)

Using the **–v***device*[*:revision*] option specifies the target for which instructions should be generated. The *device* parameter is the last four digits of the TMS320C55x part number. For example, to specify the TMS320VC5510 DSP device, you would use –v5510. The –v*device* option without a *revision* number generates code that will run on all current silicon revisions for that device. However, if you specify a revision number (e.g., –v5510:1), the compiler may be able to generate more optimal code for that revision. You can use multiple –v options to specify multiple revision numbers.

If the –v option is not used, the compiler generates code that will run on all TMS320VC5510 DSP device revisions (as if you used –v5510).

The –v*device*:0 option generates code according to the device's original hardware specification.

Note that the revision specifier can be more than one digit, if necessary. A revision numbered as 1.1 would be specified as –v*device*:1.1.

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
.c	C source
.C, .cpp, .cxx, or .cc†	C++ source
.asm, .abs, or .s* (extension begins with s)	Assembly source
.obj	Object

† 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.

All source files require an extension. The conventions for filename extensions allow you to compile C/C++ files 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 on page 2-18. For information about how you can alter the way that the shell interprets and names the extensions of assembly source and object files, see Section 2.3.5 on page 2-19.

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:

```
cl55 *.c
```

2.3.4 Changing How the Shell Program Interprets Filenames (–fa, –fc, –fg, –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 –f*x* 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 asmbly, use the –fa and –fc options to force the correct interpretation:

```
cl55 –fc file.s –fa asmbly
```

You cannot use the –f 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 2-17, for more information about filename extension conventions.

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

You can use options to change how the shell program interprets filename extensions and names the extensions of the files that it creates. On the command line, the –e*x* options must precede any filenames to which they apply. You can use wildcard specifications with these options.

Select the appropriate option for the type of extension you want to specify:

–ea[**.**] *new extension*	for an assembly source file
–eo[**.**] *new extension*	for an object file
–ec[**.**] *new extension*	for a C source file
–ep[**.**] *new extension*	for a C++ source file
–es[**.**] *new extension*	for an assembly listing file

An extension can be up to nine characters in length.

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

```
cl55 –ea .rrr –eo .o fit.rrr
```

The period (.) in the extension and the space between the option and the extension are optional. The example above could be written as:

```
cl55 –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*directory* Specifies the destination directory for absolute listing files. The default is to use the same directory as the object file. To specify a listing file directory, type the directory's pathname on the command line after the -fb option:

```
cl55 –fb d:\object ...
```

–ff*directory* Specifies the destination directory for assembly listing and cross-reference listing files. The default is to use the same directory as the object file directory. Using this option without the assembly listing (-al) option or cross-reference listing (–ax) option will cause the shell to act as if the -al option was specified. To specify a listing file directory, type the directory's pathname on the command line after the –ff option:

```
cl55 –ff d:\object ...
```

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

```
cl55 –fr d:\object ...
```

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

```
cl55 –fs d:\assembly ...
```

–ft*directory* Specifies a directory for temporary intermediate files. To specify a temporary directory, insert the directory's pathname on the command line after the –ft option:

```
cl55 –ft d:\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 language source files. For example, –c makes the symbols ABC and abc equivalent. If you do not use this option, case is significant (this is the default).

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

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

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

–ar*num* Suppresses the assembler remark identified by *num*. A remark is an informational assembler message that is less severe than a warning. If you do not specify a value for *num*, all remarks will be suppressed.

–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.

–ata (ARMS mode) Informs the assembler that the ARMS status bit will be enabled during the execution of this source file. By default, the assembler assumes that the bit is disabled.

–atc (CPL mode) Informs the assembler that the CPL status bit will be enabled during the execution of this source file. This causes the assembler to enforce the use of SP-relative addressing syntax. By default, the assembler assumes that the bit is disabled.

–ath	Causes the assembler to generate faster code rather than smaller code when porting your C54x files. By default, the assembler tries to encode for small code size.
–atl	('C54x compatibility mode) Informs the assembler that the C54CM status bit will be enabled during the execution of this source file. By default, the assembler assumes that the bit is disabled.
–atn	Causes the assembler to remove NOPs located in the delay slots of C54x delayed branch/call instructions.
–att	Informs the assembler that the SST status bit will be disabled during the execution of this ported C54x source file. By default, the assembler assumes that the bit is enabled.
–atv	Causes the assembler to use the largest (P24) form of certain variable-length instructions. By default, the assembler tries to resolve all variable-length instructions to their smallest size.
–atw	Suppresses assembler warning messages. (Supported for the algebraic assembler only.)
–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.
––purecirc	Informs the assembler that the C54x file uses C54x circular addressing (does not use the C55x linear/circular mode bits).

For more information about the assembler, see the *TMS320C55x Assembly Language Tools User's Guide.*

2.4 Using Environment Variables

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

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

2.4.1 Specifying Directories (C_DIR and C55X_C_DIR)

The compiler uses the C55X_C_DIR and C_DIR environment variables to name alternate directories that contain #include files. The shell looks for the C55X_C_DIR environment variable first and then reads and processes it. If it does not find this variable, it reads the C_DIR environment variable and processes it. To specify directories for #include files, set C_DIR with one of these commands:

Operating System	Enter
Windows™	`set C_DIR=`*directory1*`[;`*directory2* ...`]`
UNIX	`setenv C_DIR "`*directory1* [*directory2* ...]`"`

The environment variable remains set until you reboot the system or reset the variable.

2.4.2 Setting Default Shell Options (C_OPTION and C55X_C_OPTION)

You might find it useful to set the compiler, assembler, and linker shell default options using the C55X_C_OPTION or C_OPTION environment variable. If you do this, the shell uses the default options and/or input filenames that you name with C_OPTION every time you run the shell.

Setting the default options with the C_OPTION environment variable is useful when you want to run the shell consecutive times with the same set of options and/or input files. After the shell reads the command line and the input filenames, it looks for the C55X_C_OPTION environment variable first and then reads and processes it. If it does not find the C55X_C_OPTION, 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*$_2$ . . .]**"**
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/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:

```
cl55 *.c              ; compiles and links
cl55 -c *.c           ; only compiles
cl55 *.c -z lnk.cmd   ; compiles/links using .cmd file
cl55 -c *.c -z lnk.cmd  ; only compiles (-c overrides -z)
```

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

2.5 Controlling the Preprocessor

This section describes specific features that control the C55x preprocessor, which is part of the parser. A general description of C preprocessing is in Section A12 of K&R. The C55x 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
__TMS320C55X__	Always defined
__COMPILER_VERSION__	Expands to an integer value representing the current compiler version number. For example, version 1.20 is represented as 120.
__LARGE_MODEL__	Expands to 1 when the –ml shell option is specified; otherwise, it is undefined
__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 if optimization is used; undefined otherwise. Regardless of any optimization, always undefined when –pi is used.

† 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 1999");
```

2.5.2 The Search Path for #include Files

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

- If you enclose the filename in double quotes (" "), the compiler searches for the file in the following directories in this order:
  1) The directory that contains the current source file. The current source file refers to the file that is being compiled when the compiler encounters the #include directive.
  2) Directories named with the –i option
  3) Directories set with the C55X_C_DIR or C_DIR environment variables
- If you enclose the filename in angle brackets (< >), the compiler searches for the file in the following directories in this order:
  1) Directories named with the –i option
  2) Directories set with the C55X_C_DIR or C_DIR environment variables

See Section 2.5.2.1, *Changing the #include File Search Path With the –i Option*, for information on using the –i option. For information on how to use the C_DIR environment variable, see Section 2.4.1, *Specifying Directories (C_DIR and C55X._C_DIR)*.

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

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

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

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:\tools\files\alt.h
UNIX	/tools/files/alt.h

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

Operating System	Enter
Windows	`cl55 –ic:\tools\files source.c`
UNIX	`cl55 –i/tools/files source.c`

2.5.3 Generating a Preprocessed Listing File (–ppo Option)

The –ppo option allows you to generate a preprocessed version of your source file. The preprocessed file has the same name as the source file but with a .pp extension. 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.c***", line** *n***:** *diagnostic severity***:** *diagnostic message*

"*file.c***"**	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/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 what the error refers to.

2.6.1 Controlling Diagnostics

The 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 Produces diagnostics information file with the same name as the corresponding source file but with an *.err* extension.

–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.c", line 9: warning: statement is unreachable
"err.c", 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.c]
"err.c", line 9: warning #112-D: statement is unreachable
"err.c", line 12: warning #112-D: statement is unreachable
```

Next, you can use the diagnostic identifier of 112 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.

For example:

```
cl55 –j
>> invalid option –j (ignored)
>> no source files
```

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

The –px shell option generates a cross-reference listing file (.crl) 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 shell 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 (.rl) that can help you understand how the compiler is preprocessing your source file. Whereas the preprocessed listing file (generated with the –ppo, –ppc, and –ppl 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 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 expands 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 C55x. The compiler replaces intrinsic operators with efficient code (usually one instruction). This “inlining” happens automatically whether or not you use the optimizer.

For details about intrinsics, and a list of the intrinsics, see Section 6.5.4, *Using Intrinsics to Access Assembly Language Statements*, on page 6-24.

Additional functions that may be expanded inline are:

- ❏ abs
- ❏ labs
- ❏ fabs
- ❏ _assert
- ❏ _nassert
- ❏ memcpy

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 3.6, *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 the strlen function. The first, in the header file, is an inline definition. This definition is enabled and the prototype is declared as static inline only if _INLINE is true (_INLINE is automatically defined for you when the optimizer is used and –pi is not specified).

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

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

(a) string.h

```
/*****************************************************************************/
/* string.h   v x.xx                                                         */
/* Copyright (c) 2001 Texas Instruments Incorporated                         */
/*****************************************************************************/

; . . .
#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned int size_t;
#endif

#ifdef _INLINE
#define __INLINE static inline
#else
#define __INLINE
#endif

__INLINE size_t  strlen(const char *_string);
; . . .

#ifdef _INLINE

/*****************************************************************************/
/*  strlen                                                                   */
/*****************************************************************************/
static inline size_t strlen(const char *string)
{
   size_t       n = (size_t) –1;
   const char *s = string – 1;
   do n++; while (*++s);
   return n;
}
; . . .

#endif
#undef __INLINE
#endif
```

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

(b) strlen.c

```
/*****************************************************************************/
/*  strlen v x.xx                                                            */
/*  Copyright (c) 2001  Texas Instruments Incorporated                       */
/*****************************************************************************/
#undef _INLINE
#include <string.h>

size_t strlen(const char *string)
{
   size_t n = (size_t) –1;
   const char *s = string – 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 declared as static inline. In functions declared as static inline, expansion occurs despite the presence of local static variables. In addition, a limit is placed on the depth of inlining for recursive or nonleaf functions. 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 –ss option. To compile and run the interlist utility on a program called function.c, enter:

```
cl55 –ss function
```

The –ss 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. For more information about using the interlist utility with the optimizer, see Section 3.7, *Using the Interlist Utility With the Optimizer*, on page 3-13.

Example 2–3. An Interlisted Assembly Language File

```
       .global   _main
;-------------------------------------------------------------------------
;   3 | void main (void)
;-------------------------------------------------------------------------
;*************************************************************************
;* FUNCTION NAME       : _main                                           *
;* Function Uses Regs  : M40,SATA,SATD,FRCT,SMUL                         *
;* Stack Frame         : Compact (No Frame Pointer, w/ debug)            *
;* Total Frame Size    : 3 words                                         *
;*                      (1 return address/alignment)                    *
;*                        (2 function parameters)                       *
;*************************************************************************
_main:
        AADD #-3, SP
;-------------------------------------------------------------------------
;   5 | printf("Hello World\n");
;-------------------------------------------------------------------------
        MOV #SL1, *SP(#0)
        call      #_printf               ;
                                        ; call occurs [#_printf]   ;
        AADD #3, SP
        return                          ; return occurs
;*************************************************************
;* STRINGS                                                   *
;*************************************************************
       .sect  ".const"
       .align 1
SL1:   .string   "Hello World",10,0
;*************************************************************
;* UNDEFINED EXTERNAL REFERENCES                             *
;*************************************************************
   .global   _printf
```

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

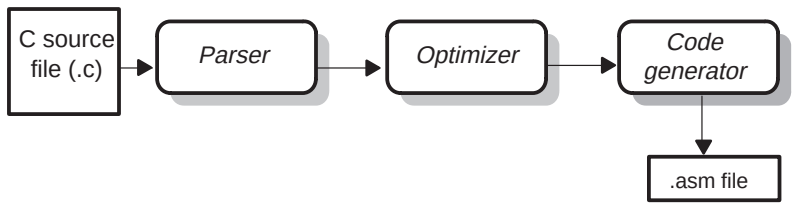
Topic **Page**

3.1 Using the Optimizer

The C/C++ compiler is able to perform various optimizations. High-level optimizations are performed in the optimizer and low-level, target-specific optimizations occur in the code generator.

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

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



The easiest way to invoke the optimizer is to use the cl55 shell program, specifying the –o*n* option on the cl55 command line. The *n* denotes the level of optimization (0, 1, 2, and 3), which controls the type and degree of optimization:

❑ **–o0**

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

❑ **–o1**

Performs all –o0 optimizations, plus:

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

❑ **–o2**

Performs all –o1 optimizations, plus:

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

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

❑ **–o3**

Performs all –o2 optimizations, plus:

- Removes all functions that are never called
- Simplifies functions with return values that are never used
- Inlines calls to small functions
- Reorders function declarations so that the attributes of called functions are known when the caller is optimized
- Identifies file-level variable characteristics

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

The levels of optimization described above are performed by the stand-alone optimization pass. The code generator performs several additional optimizations, particularly processor-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 (–o3 Option)

The –o3 option instructs the compiler to perform file-level optimization. You can use the –o3 option alone to perform general file-level optimization, or you can combine it with other options to perform more specific optimization. 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 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 –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 and alters it	–ol0
Contains definitions of library functions that are identical to the standard library functions; does not alter functions declared in the standard library	–ol1
Does not alter standard library functions, but you used the –ol0 or the –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 Option)

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

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

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

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

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

- If a particular argument in a function always has the same value, the compiler replaces the argument with the value and passes the value instead of the argument.
- If a return value of a function is never used, the compiler deletes the return code in the function.
- If a function is not called directly or indirectly by main, 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 subsection 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 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 the 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 definitions 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 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 in other modules	–op2
Has functions that are called from other modules but does not have global variables that are modified in other modules	–op3

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

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

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

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

3.3.2 Optimization Considerations When Mixing C and Assembly

If you have any assembly functions in your program, you need to exercise caution when using the –pm option. The compiler recognizes only the C/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. To keep these functions, place the FUNC_EXT_CALLED pragma (see subsection 5.7.6, *The FUNC_EXT_CALLED Pragma*, on page 5-21) before any declaration or reference to a function that you want to keep.

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

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

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

Situation Your application consists of C/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. See subsection 3.3.1 for information about the –op2 option.

If you compile with the –pm –o3 options only, the compiler reverts from the default optimization level (–op2) to –op0. The compiler uses –op0, because it presumes that the calls to the assembly language functions that have a definition in C/C++ may 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. (See subsection 5.4.6, *The volatile Keyword*, page 5-13, for more information.)

See subsection 3.3.1 for information about the –op option.

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

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

- You achieve the best optimization by applying the FUNC_EXT_CALLED pragma to all of the entry-point functions called from the assembly language interrupts, and then compiling with –pm –o3 –op2. *Ensure 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.

See subsection 5.7.6, on page 5-21 for information about the FUNC_EXT_CALLED pragma and subsection 3.3.1 for information about the –op option.

3.4 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 where the assembly code is inserted can differ significantly from the original C/C++ source code. It is usually safe to use asm statements to manipulate hardware controls such as interrupt masks, but asm statements that attempt to interface with the C/C++ environment or access C/C++ variables can have unexpected results. After compilation, check the assembly output to make sure your asm statements are correct and maintain the integrity of the program.

3.5 Accessing Aliased Variables in Optimized Code

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 refer to another object. The optimizer analyzes the code to determine where aliasing can and cannot occur, then optimizes as much as possible while still preserving the correctness of the program. The optimizer behaves conservatively. If there is a chance that two pointers are pointing at the same object, then the optimizer assumes that the pointers do point to the same object.

The optimizer assumes that any variable whose address is passed as an argument to a function is not 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 variable

If you use aliases like this, you must use the –ma shell 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;
}
```

3.6 Automatic Inline Expansion (–oi Option)

The optimizer automatically inlines small functions when it is invoked with the –o3 option. A command-line option, –oi*size*, specifies the size threshold. Any function larger than the *size* threshold will not be 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 the *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 option, the optimizer inlines very small functions.

Note: –o3 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. These options cause the compiler to inline intrinsics only.

3.7 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 also.

The interlist utility can affect optimized code because it might 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. These statements may not reflect the exact C/C++ syntax of the operation being performed.

Example 3–1 shows the function from Example 2–3 on page 2-43 compiled with the optimizer (–o2) and the –os option. Note that the assembly file contains optimizer comments interlisted with assembly code.

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

```
_main:
;** 5  ------------------ printf((char *)”Hello, world\n”);
;** 5  ------------------ return;
       AADD #-3, SP
       MOV #SL1, *SP(#0)
       call #_printf
                    ; call occurs [#_printf]
       AADD #3, SP
       return              ;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 2–3 on page 2-43 compiled with the optimizer (–o2) and the –ss and –os options. Note that the assembly file contains optimizer comments and C source interlisted with assembly code.

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

```
_main:
;** 5  -----------------  printf((char *)"Hello, world\n");
;**    -----------------   return;
          AADD #-3, SP
;----------------------------------------------------------------------------
;   5 | printf("Hello, world\n");
;----------------------------------------------------------------------------
          MOV #SL1, *SP(#0)
          call #_printf
                        ; call occurs [#_printf]
          AADD #3, SP
          return              ;return occurs
```

3.8 Debugging Optimized Code

Debugging fully 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 alleviate this problem, you have two alternatives:

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

- Use the –g, –o, and –mn options. The –mn option reenables the optimizations disabled by –g, making your optimized code slightly easier to debug than if you just use –g and –o.

With either alternative, portions of the debugger's functionality will be unreliable.

3.9 What Kind of Optimization Is Being Performed?

The TMS320C55x™ 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 that you cannot selectively enable or disable.

Following are the optimizations performed by the compiler.

Optimization	Page
Cost-based register allocation	3-17
Alias disambiguation	3-17
Branch optimizations and control-flow simplification	3-17
Data flow optimizations ❑ Copy propagation ❑ Common subexpression elimination ❑ Redundant assignment elimination	3-19
Expression simplification	3-19
Inline expansion of runtime-support library functions	3-21
Induction variable optimizations and strength reduction	3-22
Loop-invariant code motion	3-22
Loop rotation	3-22

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

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 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 Branch Optimizations and Control-Flow Simplification

The compiler analyzes the branching behavior of a program and rearranges the linear sequences of operations (basic blocks) to remove branches or redundant conditions. Unreachable code is deleted, branches to branches are bypassed, and conditional branches over unconditional branches are simplified to a single conditional branch.

When the value of a condition is determined at compile time (through copy propagation or other data flow analysis), the compiler can delete a conditional branch. Switch case lists are analyzed in the same way as conditional branches and are sometimes eliminated entirely. Some simple control flow constructs can be reduced to conditional instructions, totally eliminating the need for a branch.

In Example 3–3, the switch statement and the state variable from this simple finite state machine example are optimized completely away, leaving a streamlined series of conditional branches.

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

(a) C source

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

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

(b) C compiler output

```
; opt55 –o3 control.if control.opt
;*************************************************************
;* FUNCTION NAME: _fsm                                       *
;*************************************************************
_fsm:
       MOV *AR3+, AR1
       BCC L2, AR1!=#0
L1:
       MOV *AR3+, AR1
       BCC L2, AR1==#0
       MOV *AR3+, AR1
       BCC L1, AR1==#0
L2:
       MOV *AR3+,AR1
       BCC L2, AR1==#0

       return
```

3.9.4 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 3–3 on page 3-18 and Example 3–4 on page 3-20.

- **Common subexpression elimination**

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

- **Redundant assignment elimination**

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

3.9.5 Expression Simplification

For optimal evaluation, the compiler simplifies expressions into equivalent forms, requiring fewer instructions or registers. Operations between constants are folded into single constants. For example, a = (b + 4) – (c + 1) becomes a = b – c + 3.

In Example 3–4, the constant 3, assigned to a, is copy propagated to all uses of a; a becomes a dead variable and is eliminated. The sum of multiplying j by 3 plus multiplying j by 2 is simplified into b = j * 5. The assignments to a and b are eliminated.

Example 3–4. Data Flow Optimizations and Expression Simplification

(a) C source

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

(b) C compiler output

```
; opt55 –o2 data.if data.opt
;*************************************************************
;* FUNCTION NAME: _simplify                                  *
;*************************************************************
_simplify:
       MOV T0, HI(AC0)
       MPYK #5,AC0,AC0
       MOV AC0, T0
       return
```

3.9.6 Inline Expansion of Functions

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

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

Example 3–5. Inline Function Expansion

(a) C source

```
int plus (int x, int y)
{
   return x + y;
}
main ()
{
   int a = 3;
   int b = 4;
   int c = 5;

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

(b) C compiler output

```
; opt55 –o3 inline.if inline.opt
   .sect ".text"
   .global _plus
;***************************************************
;* FUNCTION NAME: _plus                            *
;***************************************************
_plus:
   ADD T0,T1
   MOV T1,T0
   return

   .sect ".text"
   .global _main
;***************************************************
;* FUNCTION NAME: _main                            *
;***************************************************
_main:

; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> ENTERING plus()
; <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< LEAVING plus()
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> ENTERING plus()
; <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< LEAVING plus()
   MOV #12,T0
   return
```

3.9.7 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 entirely.

3.9.8 Loop-Invariant Code Motion

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

3.9.9 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.

Chapter 4

Linking C/C++ Code

The TMS320C55x™ C/C++ compiler and assembly language tools provide two methods for linking your programs:

- ❏ You can compile individual modules and then link them together. This method is especially useful when you have multiple source files.
- ❏ You can compile and link in one step by using cl55. This method is useful when you have a single source module.

This chapter describes how to invoke the linker with each method. It also discusses special requirements of linking C/C++ code, including the runtime-support libraries, specifying the initialization model, and allocating the program into memory. For a complete description of the linker, see the *TMS320C55x Assembly Language Tools User's Guide*.

Topic **Page**

4.1 Invoking the Linker as an Individual Program

The examples in this section show 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:

lnk55 {**–c**|**–cr**} *filenames* [*options*] [**–o** *name*.out] –l *library* [*lnk.cmd*]

lnk55 — 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 lnk55, you must use –c or –cr. The –c option uses automatic variable initialization at runtime; the –cr option uses automatic variable initialization at reset.

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.

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

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

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.obj, prog2.obj, and prog3.obj (the output file is named prog.out), enter:

```
lnk55 –c prog1 prog2 prog3 –o prog.out –l rts55.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 *TMS320C55x 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 are linking with the shell, the options follow the –z shell option (see subsection 2.2, *Invoking the C Compiler Shell*, on page 2-4).

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

- The –z option divides the command line into compiler options (the options before –z) and linker options (the options following –z).
- The –z option must follow all source files and 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:

```
cl55 –sq *.c –z c.cmd –o prog.out –l rts55.lib
```

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

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

1) Object filenames from the command line
2) Arguments following the –z option on the command line
3) Arguments following the –z option from the C_OPTION or C55X_C_OPTION environment variable

4.3 Disabling the Linker (–c Shell Option)

You can override the –z option by using the –c option. The –c option is especially helpful if you specify the –z option in the C_OPTION or C55X_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 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 runtime). If you want to autoinitialize variables at reset, 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 detailed descriptions of their effects:

–a	Produces an absolute, executable module. This is the default; if neither –a nor –r is specified, the linker acts as if –a is specified.
–ar	Produces a relocatable, executable object module
–b	Disables merge of symbolic debugging information
–c	Autoinitializes variables at runtime
–cr	Autoinitializes variables at reset
–e *global_symbol*	Defines a *global_symbol* that specifies the primary entry point for the output module
–f *fill_value*	Sets the default fill value for holes within output sections; *fill_value* is a 16-bit constant
–g *global_symbol*	Defines a *global_symbol* as global even if the global symbol has been made static with the –h linker option
–h	Makes all global symbols static
–heap *size*	Sets heap size (for the dynamic memory allocation) to *size* bytes and defines a global symbol that specifies the heap size. The default is 2000 bytes.
–i *directory*	Alters the library-search algorithm to look in *directory* before looking in the default location. This option must appear before the –l linker option. The directory must follow operating system conventions.
–j	Disables conditional linking
–k	Ignore alignment flags in input sections
–l *filename*	(lowercase L) Names an archive library file as linker input; *filename* is an archive library name and must follow operating system conventions.
–m *filename*	Produces a map or listing of the input and output sections, including holes, and places the listing in *filename.* The filename must follow operating system conventions.

–o *filename*	Names the executable output module. The *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 primary C/C++ system stack size to *size* bytes and defines a global symbol that specifies the stack size. The default is 1000 bytes.
–sysstack *size*	Sets the secondary system stack size to *size* bytes and defines a global symbol that specifies the secondary stack size. The default is 1000 bytes.
–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. Resolves back references.

For more information on linker options, see the *Linker Description* chapter of the *TMS320C55x 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 runtime-support library
- ❏ Specify the initialization model
- ❏ Determine how you want to allocate your program into memory

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

For more information about how to operate the linker, see the *Linker Description* chapter of the *TMS320C55x Assembly Language Tools User's Guide.*

4.5.1 Linking With Runtime-Support Libraries

You must link all C/C++ programs with a runtime-support library. The library contains standard C/C++ functions as well as functions used by the compiler to manage the C/C++ environment. You must use the –l linker option to specify the runtime-support library to use. The –l option also tells the linker to look at the –i options and then the C_DIR environment variable to find an archive path or object file.

To use the –l option, type on the command line:

lnk55 {**–c** | **–cr**} *filenames* **–l** *libraryname*

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

4.5.2 Runtime 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 for initializing the runtime environment; the linker automatically extracts boot.obj and links it when you use –c or –cr and include the appropriate runtime library in the link.

The rts55.lib and rts55x.lib archive libraries contain C/C++ runtime-support functions.

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

- Sets up the stack and secondary stack
- Processes the runtime initialization table and autoinitializes global variables (when using the –c option)
- Calls all global constructors
- Calls main
- Calls exit when main returns

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

Note: The _c_int00 Symbol

One important function contained in the runtime 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 reset, you should set up the reset vector to branch to _c_int00 so that the processor executes boot.obj first.

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.

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 and before main() is called. Destructors are registered through the atexit() system call and therefore are invoked during the call to exit().

Section 6.8.3, *Initialization Tables*, on page 6-37 discusses the format of the .pinit table.

4.5.4 Specifying the Type of Initialization

The C/C++ compiler produces data tables for autoinitializing global variables. Subsection 6.8.3, *Initialization Tables*, on page 6-37 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 runtime. Global variables are initialized at *runtime*. Use the –c linker option (see subsection 6.8.4, *Autoinitialization of Variables at Runtime*, on page 6-41).
- Autoinitializing variables at load time. Global variables are initialized at *load time*. Use the –cr linker option (see subsection 6.8.5, *Autoinitialization of Variables at Reset*, on page 6-42).

When you link a C/C++ program, you must use either the –c or the –cr option. These options tell the linker to select autoinitialization at runtime or reset. When you compile and link programs, the –c linker option is the default; if used, the –c linker option must follow the –z option (see Section 4.2, *Invoking the Linker With the Compiler Shell (–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; this ensures that boot.obj is automatically linked in from the runtime-support library.
- The *.cinit* output section is padded with a termination record so that the loader (reset initialization) or the boot routine (runtime initialization) knows when to stop reading the initialization tables.
- When autoinitializing 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 runtime.
  - The STYP_COPY flag 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 runtime (–c 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.

Note: Boot Loader

Note that a loader is not included as part of the C/C++ compiler tools. Use the C55x simulator or emulator with the source debugger as a loader.

4.5.5 Specifying Where to Allocate Sections in Memory

The compiler produces 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.

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

Table 4–1. Sections Created by the Compiler

(a) Initialized sections

Name	Contents
.cinit	Tables for explicitly initialized global and static variables
.const	Global and static const variables that are explicitly initialized and string literals
.text	Executable code
.switch	Switch statement tables

(b) Uninitialized sections

Name	Contents
.bss	Global and static variables
.cio	C I/O buffer
.stack	Primary stack
.sysstack	Secondary system stack
.sysmem	Memory for malloc functions

When you link your program, you must specify where to allocate the sections in memory.

Note:

When allocating sections, keep in mind that the .stack and .sysstack sections must be on the same page. Also, only code sections are allowed to cross page boundaries.

In general, initialized sections are linked into ROM or RAM; uninitialized sections are linked into RAM. See subsection 6.1.3, *Sections*, on page 6-3 for a complete description of how the compiler uses these sections. The linker provides MEMORY and SECTIONS directives for allocating sections. For more information about allocating sections into memory, see the *Linker Description* chapter of the *TMS320C55x 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/C++ program. The command file in this example is named lnk.cmd and lists several linker options. To link the program, enter:

lnk55 *object_file(s)* **–o** *outfile* **–m** *mapfile* lnk.cmd

The MEMORY and possibly the SECTIONS directive might require modification to work with your system. See the *Linker Description* chapter of the *TMS320C55x Assembly Language Tools User's Guide* for information on these directives.

Example 4–1. Linker Command File

```
–stack  0x2000      /* PRIMANY STACK SIZE            */
–sysstack  0x1000   /* SECONDARY STACK SIZE          */
–heap   0x2000      /* HEAP AREA SIZE                */
–c                 /* Use C linking conventions: auto–init vars at runtime*/
–u _Reset           /* Force load of reset interrupt handler  */


/* SPECIFY THE SYSTEM MEMORY MAP */
MEMORY
{
 PAGE 0:  /* ––– Unified Program/Data Address Space ––– */
   RAM (RWIX): origin = 0x000100, length = 0x01FF00 /* 128Kb page of RAM */
   ROM (RIX) : origin = 0x020100, length = 0x01FF00 /* 128Kb page of ROM */
   VECS (RIX): origin = 0xFFFF00, length = 0x000100 /* 256–byte int vector */

 PAGE 1: /* ––––– 64K–word I/O Address Space ––––––– */
   IOPORT (RWI) : origin = 0x000000, length = 0x020000
}
/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
   .text      > ROM  PAGE 0    /* CODE                          */
 /* These sections must be on same physical memory page when */
 /* small memory model is used                                */
   .data      > RAM  PAGE 0    /* INITIALIZED VARS              */
   .bss       > RAM  PAGE 0    /* GLOBAL & STATIC VARS          */
   .const     > RAM  PAGE 0    /* CONSTANT DATA                 */
   .sysmem    > RAM  PAGE 0    /* DYNAMIC MEMORY (malloc)       */
   .stack     > RAM  PAGE 0    /* PRIMARY SYSTEM STACK          */
   .sysstack  > RAM  PAGE 0    /* SECONDARY SYSTEM STACK        */
   .cio       > RAM  PAGE 0    /* C I/O BUFFERS                 */
 /* The .switch, .cinit, and .pinit sections may be on any   */
 /* physical memory page when small memory model is used     */
   .switch    > RAM  PAGE 0    /* SWITCH STATEMENT TABLES       */
   .cinit     > RAM  PAGE 0    /* AUTOINITIALIZATION TABLES     */
   .pinit     > RAM  PAGE 0    /* INITIALIZATION FN TABLES      */

   .vectors   > VECS PAGE 0    /* INTERRUPT VECTORS             */

   .ioport    > IOPORT PAGE 1  /* GLOBAL & STATIC IO VARS       */
}
```

Chapter 5

TMS320C55x C/C++ Language

The TMS320C55x™ C/C++ compiler supports the C/C++ language standard that was developed by a committee of the American National Standards Institute (ANSI) to standardize the C programming language.

The C++ language supported by the C55x is defined in *The Annotated C++ Reference Manual* (ARM). In addition, many of the extensions from the ISO/IEC 14882–1998 C++ standard are implemented.

Topic **Page**

5.1 Characteristics of TMS320C55x C

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

The ANSI standard identifies certain 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 can differ among standard compilers. This section describes how these features are implemented for the C55x C/C++ compiler.

The following list identifies all such cases and describes the behavior of the C55x C/C++ compiler in each case. Each description also includes a reference to more information. Many of the references are to the formal ANSI standard for C or to the second edition of *The C Programming Language* by Kernighan and Ritchie (K&R).

5.1.1 Identifiers and Constants

- The first 100 characters of all identifiers are significant. Case is significant; uppercase and lowercase characters are distinct for identifiers. These characteristics apply to all identifiers, internal and external.
  (ANSI 3.1.2, K&R A2.3)

- The source (host) and execution (target) character sets are assumed to be ASCII. There are no multibyte characters.
  (ANSI 2.2.1, K&R A12.1)

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

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

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

5.1 Characteristics of TMS320C55x C

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

The ANSI standard identifies certain 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 can differ among standard compilers. This section describes how these features are implemented for the C55x C/C++ compiler.

The following list identifies all such cases and describes the behavior of the C55x C/C++ compiler in each case. Each description also includes a reference to more information. Many of the references are to the formal ANSI standard for C or to the second edition of *The C Programming Language* by Kernighan and Ritchie (K&R).

5.1.1 Identifiers and Constants

- The first 100 characters of all identifiers are significant. Case is significant; uppercase and lowercase characters are distinct for identifiers. These characteristics apply to all identifiers, internal and external.
  (ANSI 3.1.2, K&R A2.3)

- The source (host) and execution (target) character sets are assumed to be ASCII. There are no multibyte characters.
  (ANSI 2.2.1, K&R A12.1)

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

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

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

5.1.2 Data Types

- For information about the representation of data types, see Section 5.3.

  (ANSI 3.1.2.5, K&R A4.2)
- The type size_t, which is the result of the *sizeof* operator, is unsigned int. (ANSI 3.3.3.4, K&R A7.4.8)
- The type ptrdiff_t, which is the result of pointer subtraction, is int. (ANSI 3.3.6, K&R A7.7)

5.1.3 Conversions

- Float-to-integer conversions truncate toward 0. (ANSI 3.2.1.3, K&R A6.3)
- Pointers and integers can be freely converted, as long as the result type is large enough to hold the original value. (ANSI 3.3.4, K&R A6.6)

5.1.4 Expressions

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

  ```
  10 / –3 == –3,     –10 / 3 == –3
  10 % –3 == 1,      –10 % 3 == –1
  ```
  (ANSI 3.3.5, K&R A7.6)

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

5.1.5 Declaration

- The *register* storage class is effective for all chars, shorts, ints, and pointer types. (ANSI 3.5.1, K&R A2.1)
- Structure members are packed into words. (ANSI 3.5.2.1, K&R A8.3)
- A bit field defined as an integer is signed. Bit fields are packed into words and do not cross word boundaries. (ANSI 3.5.2.1, K&R A8.3)
- The interrupt keyword can be applied only to void functions that have no arguments. For more information, see subsection 5.4.3 on page 5-11. (TI C extension)

5.1.6 Preprocessor

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

The following pragmas *are* supported.

- CODE_SECTION
- C54X_CALL
- C54X_FAR_CALL
- DATA_ALIGN
- DATA_SECTION
- FUNC_CANNOT_INLINE
- FUNC_EXT_CALLED
- FUNC_IS_PURE
- FUNC_IS_SYSTEM
- FUNC_NEVER_RETURNS
- FUNC_NO_GLOBAL_ASG
- FUNC_NO_IND_ASG
- MUST_ITERATE
- UNROLL

For more information on pragmas, see Section 5.7 on page 5-16.

5.2 Characteristics of TMS320C55x C++

The C55x compiler supports C++ as defined in Ellis and Stroustrup's *The Annotated C++ Reference Manual* (ARM). In addition, many of the features of the ISO/IEC 14882–1998 C++ standard are accepted. The exceptions to the standard are as follows:

- Complete C++ standard library support is not included. In particular, the iostream library is not supported. C subset and basic language support is included.
- Exception handling is not supported.
- Run-time type information (RTTI) is disabled by default. RTTI allows the type of an object to be determined at run time. It can be enabled with the –rtti shell option.
- The only C++ standard library header files included are <typeinfo> and <new>. Support for bad_cast or bad_type_id is not included in the typeinfo header.
- The following C++ headers for C library facilities are not included:
  - <ciso646>
  - <clocale>
  - <csignal>
  - <cwchar>
  - <cwctype>
- The reinterpret_cast type does not allow casting a pointer-to-member of one class to a pointer-to-member of 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 parameters are not implemented.
- The export keyword for templates is not implemented.
- A partial specialization of a class member template cannot be added outside of the class definition.

5.3 Data Types

Table 5–1 lists the size, representation, and range of each scalar data type or the C55x compiler. Many of the range values are available as standard macros in the header file limits.h. For more information, see subsection 7.3.5, *Limits (float.h and limits.h)*, on page 7-16.

Table 5–1. TMS320C55x C/C++ Data Types

Type	Size	Representation	Minimum Value	Maximum Value
signed char	16 bits	ASCII	–32 768	32 767
char, unsigned char	16 bits	ASCII	0	65 535
short, signed short	16 bits	2s complement	–32 768	32 767
unsigned short	16 bits	Binary	0	65 535
int, signed int	16 bits	2s complement	–32 768	32 767
unsigned int	16 bits	Binary	0	65 535
long, signed long	32 bits	2s complement	–2 147 483 648	2 147 483 647
unsigned long	32 bits	Binary	0	4 294 967 295
long long	40 bits	2s complement	–549 755 813 888	549 755 813 887
unsigned long long	40 bits	Binary	0	1 099 511 627 775
enum	16 bits	2s complement	–32 768	32 767
float	32 bits	IEEE 32-bit	1.175 494e–38	3.40 282 346e+38
double	32 bits	IEEE 32-bit	1.175 494e–38	3.40 282 346e+38
long double	32 bits	IEEE 32-bit	1.175 494e–38	3.40 282 346e+38
pointers (data) small memory mode large memory mode	 16 bits 23 bits	 Binary	 0	 0xFFFF 0x7FFFFF
pointers (function)	24 bits	Binary	0	0xFFFFFF

Note: long long is 40 bits

The long long data type is implemented according to the ISO/IEC 9899 C Standard. However, the C55x compiler implements this data type as 40 bits instead of 64 bits. Use the “ll” length modifier with formatted I/O functions (such as printf and scanf) to print or read long long variables. For example:

```
printf(”%lld\n”, (long long)global);
```

5.4 Keywords

The C55x C/C++ compiler supports the standard const and volatile keywords. In addition, the C55x C/C++ compiler extends the C/C++ language through the support of the interrupt, ioport, and restrict keywords.

5.4.1 The const Keyword

The C55x C/C++ compiler supports the ANSI standard keyword *const*. This keyword gives you greater 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 their values are not altered.

If you define an object as const, the const section allocates storage for the object. The const data storage allocation rule has two exceptions:

- If the keyword volatile is also specified in the definition of an object (for example, volatile const int x). Volatile keywords are assumed to be allocated to RAM. (The program does not modify a const volatile object, but something external to the program might.)
- If the object is auto (function scope).

In both cases, the storage for the object is the same as if the const keyword were not used.

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;
```

Using the const keyword, you can define large constant tables and allocate them into system ROM. For example, to allocate a ROM table, you could use the following definition:

```
const int digits [] = {0,1,2,3,4,5,6,7,8,9};
```

5.4.2 The ioport Keyword

The C55x processor contains a secondary memory space for I/O. The compiler extends the C/C++ language by adding the ioport keyword to support the I/O addressing mode.

The ioport type qualifier may be used with the standard type specifiers including arrays, structures, unions, and enumerations. It can also be used with the const and volatile type qualifiers. When used with an array, ioport qualifies the elements of the array, not the array type itself. Members of structures cannot be qualified with ioport unless they are pointers to ioport data.

The ioport type qualifier can only be applied to global or static variables. Local variables cannot be qualified with ioport unless the variable is a pointer declaration. For example:

```
void foo (void)
{
   ioport int i;  /* invalid */
   ioport int *j;  /* valid */
```

When declaring a pointer qualified with ioport, note that the meaning of the declaration will be different depending on where the qualifier is placed. Because I/O space is 16-bit addressable, pointers to I/O space are always 16 bits, even in the large memory model.

Note that you cannot use printf() with a direct ioport pointer argument. Instead, pointer arguments in printf() must be converted to “void *”, as shown in the example below:

```
ioport int *p;
printf(”%p\n”, (void*)p);
```

The declaration shown in Example 5–1 places a pointer in I/O space that points to an object in data memory.

Example 5–1. Declaring a Pointer in I/O Space

(a) C source

```
int * ioport ioport_pointer; /* ioport pointer */
int i;
int j;

void foo (void)
{
   ioport_pointer = &i;
   j = *ioport_pointer;
}
```

Example 5–1. Declaring a Pointer in I/O Space (Continued)

(b) Compiler output

```
_foo:
    MOV#_i,port(#_ioport_pointer) ; store addr of #i
                                      ; (I/O memory)
    MOVport(#_ioport_pointer),AR3 ; load address of #i
                                      ; (I/O memory)
    MOV *AR3,AR1            ; indirectly load value of #i
    MOV AR1,*abs16(#_j)     ; store value of #i at #j
    return
```

If typedef had been used in Example 5–1, the pointer declaration would have been:

```
typedef int *int_pointer;
ioport int_pointer ioport_pointer; /* ioport pointer */
```

Example 5–2 declares a pointer that points to data in I/O space. This pointer is 16 bits even in the large memory model.

Example 5–2. Declaring a Pointer That Points to Data in I/O Space

(a) C source

```
/* pointer to ioport data: */
ioport int * ptr_to_ioport;
ioport int i;

void foo (void)
{
    int j;
    i = 10;
    ptr_to_ioport = &i;
    j = *ptr_to_ioport;
}
```

(b) Compiler output

```
_foo:
    MOV#_i,*abs16(#_ptr_to_ioport)   ; store address of #i
    MOV *abs16(#_ptr_to_ioport),AR3
    AADD #–1, SP
    MOV #10,port(#_i)       ; store 10 at #i (I/O memory)
    MOV *AR3,AR1
    MOV AR1,*SP(#0)
    AADD #1,SP
    return
```

Example 5–3 declares an ioport pointer that points to data in I/O space.

Example 5–3. Declaring an ioport Pointer That Points to Data in I/O Space

(a) C source

```
/* ioport pointer to ioport data: */
ioport int * ioport iop_ptr_to_ioport;
ioport int i;
ioport int j;

void foo (void)
{
   i = 10;
   iop_ptr_to_ioport = &i;
   j = *iop_ptr_to_ioport;
}
```

(b) Compiler output

```
_foo:
   MOV #10,port(#_i)       ; store 10 at #i (I/O memory)
   MOV#_i,port(#_iop_ptr_to_ioport) ; store address of
                                ; #i (I/O memory)
   MOV port(#_iop_ptr_to_ioport),AR3 ; load addr of #i
MOV *AR3, AR1           ; load #i
   MOV AR1,port(#_j)       ; store 10 in #j (I/O memory)
   return
```

5.4.3 The interrupt Keyword

The C55x compiler extends the C/C++ language by adding the interrupt keyword to specify that a function is to be treated as an interrupt function.

Functions that handle interrupts require 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 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. For example:

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

    ...
}
```

The name c_int00 is the C/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.

Use the alternate keyword, __interrupt, if you are writing code for strict ANSI mode (using the –ps shell option).

5.4.4 The onchip Keyword

The onchip keyword informs the compiler that a pointer points to data that may be used as an operand to a dual MAC instruction. Data passed to a function with onchip parameters, or data that will be eventually referenced through an onchip expression, must be linked into on-chip memory (not external memory). Failure to link the data appropriately can result in a reference to external memory through the BB data bus, which will generate a bus error.

```
onchip int x[100];    /* array declaration */
onchip int *p;        /* pointer declaration */
```

The –mb shell option specifies that all data memory will be on-chip.

5.4.5 The restrict Keyword

To help the compiler determine memory dependencies, you can qualify a pointer, reference, or array with the restrict keyword. The restrict keyword is a type qualifier that may be applied to pointers, references, and arrays. Its use represents a guarantee by the programmer that within the scope of the pointer declaration the object pointed to can be accessed only by that pointer. Any violation of this guarantee renders the program undefined. This practice helps the compiler optimize certain sections of code because aliasing information can be more easily determined.

In Example 5–4, you can use the restrict keyword to tell the compiler that a and b never point to the same object in foo (and the objects' memory that foo accesses does not overlap).

Example 5–4. Use of the restrict Type Qualifier With Pointers

```
void foo(int * restrict a, int * restrict b)
{
   /* foo's code here  */
}
```

Example 5–5 illustrates using the restrict keyword when passing arrays to a function. Here, the arrays c and d should not overlap, nor should c and d point to the same array.

Example 5–5. Use of the restrict Type Qualifier With Arrays

```
void func1(int c[restrict], int d[restrict])
{
   int i;

   for(i = 0; i < 64; i++)
   {
     c[i] += d[i];
     d[i] += 1;
   }
}
```

5.4.6 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/C++ code, you *must* use the volatile keyword to identify these accesses. The compiler won't optimize out any references to volatile variables.

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

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

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

```
volatile unsigned int *ctrl
```

5.5 Register Variables and Parameters

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 6.3, *Register Conventions*, on page 6-9.

5.6 The asm Statement

The TMS320C55x C/C++ compiler can embed C55x assembly language instructions or directives directly into the assembly language output of the compiler. This capability is an extension to the C/C++ language—the *asm* statement. The asm statement provides access to hardware features that C/C++ cannot provide. The asm statement is syntactically like a call to a function named asm, with one string-constant argument:

asm("*assembler text***");**

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

```
asm("STR: .string \"abc\"");
```

The inserted code must be a legal assembly language statement. Like all assembly language statements, the line of code inside the quotes must begin with a label, a blank, a tab, or a comment (asterisk or semicolon). The compiler performs no checking on the string; if there is an error, the assembler detects it. For more information about assembly language statements, see the *TMS320C55x Assembly Language Tools User's Guide*.

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

Note: Avoid Disrupting the C/C++ Environment With asm Statements

Be 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 optimizer cannot remove asm statements, it can significantly rearrange the code order near them, possibly causing undesired results.

5.7 Pragma Directives

Pragma directives tell the compiler's preprocessor how to treat functions. The C55x C/C++ compiler supports the following pragmas:

- CODE_SECTION
- C54X_CALL
- C54X_FAR_CALL
- DATA_ALIGN
- DATA_SECTION
- FUNC_CANNOT_INLINE
- FUNC_EXT_CALLED
- FUNC_IS_PURE
- FUNC_IS_SYSTEM
- FUNC_NEVER_RETURNS
- FUNC_NO_GLOBAL_ASG
- FUNC_NO_IND_ASG
- MUST_ITERATE
- UNROLL

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 follow these rules, the compiler issues a warning.

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

When you mark a function with a pragma, you assert to the compiler that the function meets the pragma's specifications in every circumstance. If the function does not meet these specifications at all times, the compiler's behavior will be unpredictable.

5.7.1 The CODE_SECTION Pragma

The CODE_SECTION pragma allocates space for the *symbol* in a section named *section name*.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma CODE_SECTION (*"section name"***)** [**;**]

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

Example 5–6 demonstrates the use of the CODE_SECTION pragma.

Example 5–6. Using the CODE_SECTION Pragma

(a) C source file

```
#pragma CODE_SECTION(funcA, "codeA")
int funcA(int a)

{
   int i;
   return (i = a);
}
```

(b) Assembly source file

```
        .sect     "codeA"
        .global _funcA

;*****************************************************
;* FUNCTION NAME: _funcA                             *
;*****************************************************
_funcA:
        return          ;return occurs
```

5.7.2 The C54X_CALL and C54X_FAR_CALL Pragmas

The C54X_CALL and C54X_FAR_CALL pragmas provide a method for making calls from C55x C code to C54x assembly functions ported with masm55. These pragmas handle the differences in the C54x and C55x runtime environments.

The syntax for these pragmas in C is:

#pragma C54X_CALL (*asm_function***)** [**;**]

#pragma C54X_FAR_CALL (*asm_function***)** [**;**]

The syntax for these pragmas in C++ is:

#pragma C54X_CALL

#pragma C54X_FAR_CALL

The *function* is a C54x assembly function, unmodified for C55x in any way, that is known to work with the C54x C compiler. This pragma cannot be applied to a C function.

The appropriate pragma must appear before any declaration or call to the assembly function. Consequently, it should most likely be specified in a header file.

Use C54X_FAR_CALL only when calling C54x assembly functions that must be called with FCALL.

Use C54X_CALL for any other call to a C54x assembly function. This includes calls that are configured at assembly-time via the __far_mode symbol to use either CALL or FCALL.

When the compiler encounters one of these pragmas, it will:

1) Temporarily adopt the C54x calling conventions and the runtime environment required for ported C54x assembly code. For more information on the C54x runtime environment, see the *TMS320C54x Optimizing C Compiler User's Guide*.
2) Call the specified assembly function.
3) Capture the result.
4) Revert to the native C55x calling conventions and runtime environment.

These pragmas do not provide support for C54x assembly functions that call C code. In this case, you must modify the assembly code to account for the differences in the C54x and C55x runtime environments. For more information, see the *Migrating a C54x System to a C55x System* chapter in the TMS320C55x *Assembly Language Tools User's Guide*.

C54x assembly code ported for execution on C55x requires the use of 32-bit stack mode. However, the reset vector in the C55x runtime library (in vectors.asm) sets the stack to be in fast return mode. If you use the C54X_CALL or C54X_FAR_CALL pragma in your code, you must change the reset vector as follows:

1) Extract vectors.asm from rts.src. The rts.src file is located in the lib subdirectory.

```
ar55 –x rts.src vectors.asm
```

2) In vectors.asm, change the following line:

```
_Reset:  .ivec _c_int00, USE_RETA
```

to be:

```
_Reset:  .ivec _c_int00, C54X_STK
```

3) Re-assemble vectors.asm:

```
asm55 vectors.asm
```

4) Place the new file into the object and source libraries.

```
ar55 –r rts55.lib vectors.obj
ar55 –r rts55x.lib vectors.obj
ar55 –r rts.src vectors.asm
```

Note: Modifying the runtime libraries installed with Code Composer Studio

To modify the source and object libraries installed with Code Composer Studio, you must turn off the Read-only attribute for the files. Select the appropriate runtime library file in Windows Explorer, open its Properties dialog, and remove the check from the Read-only checkbox.

These pragmas cannot be used in:

- ❏ Indirect calls.
- ❏ Files compiled for the C55x large memory model (with the –ml shell option). Data pointers in the large memory model are 23 bits. When passed on the stack, the pointers occupy two words, which is incompatible with functions that expect these pointers to occupy one word.

Note that the C55x C compiler does not support the global register capability of the C54x C compiler.

5.7.3 The DATA_ALIGN Pragma

The DATA_ALIGN pragma, when specified with a *constant* of 2, aligns the *symbol* to a long word (32-bit boundary). No other alignment values are accepted.

The syntax of the pragma in C is:

#pragma DATA_ALIGN (*symbol*, *constant***) [;]**

The syntax of the pragma in C++ is:

#pragma DATA_ALIGN (*constant***) [;]**

5.7.4 The DATA_SECTION Pragma

The DATA_SECTION pragma allocates space for the *symbol* in a section named *section name*. This is useful if you have data objects that you want to link into an area separate from the .bss section.

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 5–7 demonstrates the use of the DATA_SECTION pragma.

Example 5–7. Using the DATA_SECTION Pragma

(a) C source file

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

(b) C++ source file

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

(c) Assembly source file

```
          .global _bufferA
          .bss    _bufferA,512,0,0
          .global _bufferB
_bufferB: .usect  ”my_sect”,512,0,0
```

5.7.5 The FUNC_CANNOT_INLINE Pragma

The FUNC_CANNOT_INLINE pragma instructs the compiler that the named function cannot be expanded inline. Any function named with this pragma overrides any inlining designated in any other way, such as by using the inline keyword.

The pragma must appear before any declaration or reference to the function.

The syntax of the pragma in C is:

#pragma FUNC_CANNOT_INLINE (*func***)** [**;**]

The syntax for the pragma in C++ is:

#pragma FUNC_CANNOT_INLINE [**;**]

In C, the argument *func* is the name of the function that cannot be inlined. In C++, the pragma applies to the next function declared. For more information, see Section 2.9, *Using Inline Function Expansion*, on page 2-37.

5.7.6 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/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/C++ functions or any other functions called by these C/C++ functions. These functions act as entry points into C/C++.

The pragma must appear before any declaration or reference to the function.

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 [**;**]

In C, the argument *func* is the name of the function that you do not want to be removed. In C++, the pragma applies to the next function declared.

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

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

5.7.7 The FUNC_IS_PURE Pragma

The FUNC_IS_PURE pragma specifies to the optimizer that the named function has no side effects. This allows the optimizer to do the following:

- ❏ Delete the call to the function if the function's value is not needed
- ❏ Delete duplicate functions

The pragma must appear before any declaration or reference to the function.

If you use this pragma on a function that does have side effects, the optimizer could delete these side effects.

The syntax of the pragma in C is:

#pragma FUNC_IS_PURE (*func***)** [**;**]

The syntax of the pragma in C++ is:

#pragma FUNC_IS_PURE [**;**]

In C, the argument *func* is the name of a function. In C++, the pragma applies to the next function declared.

5.7.8 The FUNC_IS_SYSTEM Pragma

The FUNC_IS_SYSTEM pragma specifies to the optimizer that the named function has the behavior defined by the ANSI standard for a function with that name.

This pragma can only be used with a function described in the ANSI standard (such as strcmp or memcpy). It allows the compiler to assume that you haven't modified the ANSI implementation of the function. The compiler can then make assumptions about the implementation. For example, it can make assumptions about the registers used by the function.

Do not use this pragma with an ANSI function that you have modified.

The pragma must appear before any declaration or reference to the function.

The syntax of the pragma in C is:

#pragma FUNC_IS_SYSTEM (*func***)** [;]

The syntax of the pragma in C++ is:

#pragma FUNC_IS_SYSTEM [;]

In C, the argument *func* is the name of the function to treat as an ANSI standard function. In C++, the pragma applies to the next function declared.

5.7.9 The FUNC_NEVER_RETURNS Pragma

The FUNC_NEVER_RETURNS pragma specifies to the optimizer that, in all circumstances, the function never returns to its caller. For example, a function that loops infinitely, calls exit(), or halts the processor will never return to its caller. When a function is marked by this pragma, the compiler will not generate a function epilog (to unwind the stack, etc.) for the function.

The pragma must appear before any declaration or reference to the function.

The syntax of the pragma in C is:

#pragma FUNC_NEVER_RETURNS (*func***)** [;]

The syntax of the pragma in C++ is:

#pragma FUNC_NEVER_RETURNS [;]

In C, the argument *func* is the name of the function that does not return. In C++, the pragma applies to the next function declared.

5.7.10 The FUNC_NO_GLOBAL_ASG Pragma

The FUNC_NO_GLOBAL_ASG pragma specifies to the optimizer that the function makes no assignments to named global variables and contains no asm statements.

The pragma must appear before any declaration or reference to the function.

The syntax of the pragma in C is:

#pragma FUNC_NO_GLOBAL_ASG (*func***)** [;]

The syntax of the pragma in C++ is:

#pragma FUNC_NO_GLOBAL_ASG [**;**]

In C, the argument *func* is the name of the function that makes no assignments. In C++, the pragma applies to the next function declared.

5.7.11 The FUNC_NO_IND_ASG Pragma

The FUNC_NO_IND_ASG pragma specifies to the optimizer that the function makes no assignments through pointers and contains no asm statements.

The pragma must appear before any declaration or reference to the function.

The syntax of the pragma in C is:

#pragma FUNC_NO_IND_ASG (*func***)** [**;**]

The syntax of the pragma in C++ is:

#pragma FUNC_NO_IND_ASG [**;**]

In C, the argument *func* is the name of the function that makes no assignments. In C++, the pragma applies to the next function declared.

5.7.12 The MUST_ITERATE Pragma

The MUST_ITERATE pragma specifies to the compiler certain properties of a loop. Through the use of the MUST_ITERATE pragma, you can guarantee that a loop executes a certain number of times. The information provided by this pragma helps the compiler to determine if it can generate a hardware loop (localrepeat or blockrepeat) for the loop. The pragma can also help the compiler eliminate unnecessary code.

In general, the compiler cannot use a hardware loop (localrepeat or blockrepeat) if a loop's bounds are too complicated. However, if you specify MUST_ITERATE with the exact number of the loop's iterations, the compiler may be able to use a hardware loop to increase performance and reduce code size.

This pragma is also useful if you know the minimum number of a loop's iterations. When you specify the minimum number via MUST_ITERATE, the compiler may be able to eliminate code that, in the event of 0 iterations, bypasses the hardware loop. For detailed information, see Section 5.7.12.1, *Using the*

MUST_ITERATE Pragma to Expand Compiler Knowledge of Loops, on page 5-26.

Any time the UNROLL pragma is applied to a loop, MUST_ITERATE should be applied to the same loop. In this case, the MUST_ITERATE pragma's third argument, *multiple*, should always be specified.

No statements are allowed between the MUST_ITERATE pragma and the for, while, or do-while loop to which it applies. However, other pragmas, such as UNROLL, can appear between the MUST_ITERATE pragma and the loop.

The syntax of the pragma for C and C++ is:

#pragma MUST_ITERATE (*min*, *max*, *multiple***)** [**;**]

The arguments *min* and *max* are programmer-guaranteed minimum and maximum trip counts. The trip count is the number of times a loop iterates. The trip count of the loop must be evenly divisible by *multiple*. All arguments are optional. For example, if the trip count could be 5 or greater, you can specify the argument list as follows:

```
#pragma MUST_ITERATE(5);
```

However, if the trip count could be any nonzero multiple of 5, the pragma would look like this:

```
#pragma MUST_ITERATE(5, , 5); /* A blank field for max*/
```

It is sometimes necessary for you to provide *min* and *multiple* in order for the compiler to perform unrolling. This is especially the case when the compiler cannot easily determine how many iterations the loop will perform (i.e., the loop has a complex exit condition).

When specifying a *multiple* via the MUST_ITERATE pragma, results of the program are undefined if the trip count is not evenly divisible by *multiple*. Also, results of the program are undefined if the trip count is less than the minimum or greater than the maximum specified.

If no *min* is specified, zero is used. If no *max* is specified, the largest possible number is used. If multiple MUST_ITERATE pragmas are specified for the same loop, the smallest *max* and largest *min* are used.

5.7.12.1 Using MUST_ITERATE to Expand Compiler Knowledge of Loops

Through the use of the MUST_ITERATE pragma, you can guarantee that a loop executes a certain number of times. The example below tells the compiler that the loop is guaranteed to run exactly 10 times:

```
#pragma MUST_ITERATE(10,10);
for(i = 0; i < trip_count; i++) { ...
```

In this example, if trip_count is a 16-bit variable, the compiler will generate a hardware loop even without the pragma. However, if MUST_ITERATE is not specified for a loop such as this, the compiler generates code to bypass the loop, to account for the possibility of 0 iterations. With the pragma specification, the compiler knows that the loop iterates at least once and can eliminate the loop-bypassing code.

MUST_ITERATE can specify a range for the trip count as well as a factor of the trip count. For example:

```
#pragma MUST_ITERATE(8,48,8);
for(i = 0; i < trip_count; i++) { ...
```

This example tells the compiler that the loop executes between 8 and 48 times and that the trip_count variable is a multiple of 8 (8, 16, 24, 32, 40, 48). The multiple argument allows the compiler to unroll the loop.

You should also consider using MUST_ITERATE for loops with complicated bounds. In the following example:

```
for(i2 = ipos[2]; i2 < 40; i2 += 5) { ...
```

the compiler would have to generate a divide function call to determine, at runtime, the exact number of iterations performed. The compiler will not do this. In this case, using MUST_ITERATE to specify that the loop always executes 8 times allows the compiler to generate a hardware loop:

```
#pragma MUST_ITERATE(8,8);
for(i2 = ipos[2]; i2 < 40; i2 += 5) { ...
```

5.7.13 The UNROLL Pragma

The UNROLL pragma specifies to the compiler how many times a loop should be unrolled. The optimizer must be invoked (use –o1, –o2, or –o3) in order for pragma-specified loop unrolling to take place. The compiler has the option of ignoring this pragma.

No statements are allowed between the UNROLL pragma and the for, while, or do-while loop to which it applies. However, other pragmas, such as MUST_ITERATE, can appear between the UNROLL pragma and the loop.

The syntax of the pragma for C and C++ is:

#pragma UNROLL (*n*) [;]

If possible, the compiler unrolls the loop so there are *n* copies of the original loop. The compiler only unrolls if it can determine that unrolling by a factor of *n* is safe. In order to increase the chances the loop is unrolled, the compiler needs to know certain properties:

- The loop iterates a multiple of *n* times. This information can be specified to the compiler via the *multiple* argument in the MUST_ITERATE pragma.
- The smallest possible number of iterations of the loop.
- The largest possible number of iterations of the loop.

The compiler can sometimes obtain this information itself by analyzing the code. However, the compiler can be overly conservative in its assumptions and may generate more code than is necessary when unrolling. This can also lead to not unrolling at all.

Furthermore, if the mechanism that determines when the loop should exit is complex, the compiler may not be able to determine these properties of the loop. In these cases, you must tell the compiler the properties of the loop by using the MUST_ITERATE pragma.

The following pragma specification:

```
#pragma UNROLL(1);
```

asks that the loop not be unrolled. Automatic loop unrolling also is not performed in this case.

If multiple UNROLL pragmas are specified for the same loop, it is undefined which UNROLL pragma is used, if any.

5.8 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 C functions, an underscore (_) is prefixed to the identifier name. C++ functions are prefixed with an underscore also, but the function name is modified further.

Mangling is the process of embedding a function's signature (the number and type of its parameters) into its name. Mangling occurs only in C++ code. The mangling algorithm used closely follows that described in *The Annotated Reference Manual* (ARM). Mangling allows function overloading, operator overloading, and type-safe linking.

For example, 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
```

the resulting assembly code is:

```
__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 that demangles names into those found in the original C++ source. See Chapter 9, *C++ Name Demangling*, for more information.

5.9 Initializing Static and Global Variables

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

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

```
SECTIONS
   {
      ...
      .bss: fill = 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 (but not the program).

If you burn your application into ROM, you should explicitly initialize variables that require initialization. The method demonstrated above initializes .bss to 0 only at load time, not at system reset or power up. To make these variables 0 at runtime, explicitly define them in your code.

For more information about linker command files and the SECTIONS directive, see the linker description information in the *TMS320C55x Assembly Language Tools User's Guide*.

5.9.1 Initializing Static and Global Variables With the Const Type Qualifier

Static and global variables of type *const* without explicit initializations are similar to other static and global variables because they might not be preinitialized to 0 (for the same reasons discussed in Section 5.9, *Initializing Static and Global Variables*). For example:

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

However, the initialization of const global and static variables is different because these variables are declared and initialized in a section called .const. For example:

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

corresponds to an entry in the .const section:

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

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

5.10 Changing the ANSI C 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/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 latest supported working paper. K&R C mode does not apply to C++ code.

5.10.1 Compatibility With K&R C (–pk Option)

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

There are subtle changes, however, in the language that can affect existing code. Appendix C in *The C Programming Language* (second edition, referred to in this manual as K&R) summarizes the differences between ANSI C and the first edition's previous 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 C55x ANSI C/C++ compiler, the compiler has a K&R option (–pk) that modifies some semantic rules of the language for compatibility with older code. In general, the –pk option relaxes requirements that are stricter for ANSI C than for K&R C. The –pk option does not disable any new features of the language such as function prototypes, enumerations, initializations, or preprocessor constructs. Instead, –pk simply liberalizes the ANSI rules without revoking any of the features.

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

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

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

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

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

- 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 definitions is a single definition for the object a. For most K&R compilers, this sequence is illegal, because int 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 int or unsigned. With –pk, bit fields can be legally declared with any integral type. For example:

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

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

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

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

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

5.10.2 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.

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

The compiler supports the compilation of embedded C++. In this mode, some features of C++ are removed that are of less value or too expensive to support in an embedded system. Embedded C++ omits these C++ features:

- Templates
- Exception handling
- Runtime type information
- The new cast syntax
- The keyword /mutable/
- Multiple inheritance
- Virtual inheritance

In the standard definition of embedded C++, namespaces and using-declarations are not supported. The C55x compiler nevertheless allows these features under embedded C++ because the C++ runtime support library makes use of them. Furthermore, these features impose no runtime penalty.

Chapter 6

Runtime Environment

This chapter describes the TMS320C55x™ C/C++ runtime environment. To ensure successful execution of C/C++ programs, it is critical that all runtime code maintain this environment. It is also important to follow the guidelines in this chapter if you write assembly language functions that interface with C/C++ code.

Topic **Page**

6.1 Memory

The C55x compiler treats memory as a single linear block that is partitioned into sub-blocks of code and data. Each sub-block of code or data generated by a C program is placed in its own continuous memory space. The compiler assumes that a full 24-bit address space is available in target memory.

Note: The Linker Defines the Memory Map

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

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

The compiler supports a small memory model and a large memory model. These memory models affect how data is placed in memory and accessed.

6.1.1 Small Memory Model

The use of the small memory model results in code and data sizes that are slightly smaller than when using the large memory model. However, your program must meet certain size and memory placement restrictions.

In the small memory model, the following sections must all fit within a single page of memory that is 64K words in size:

- .bss and .data sections (all static and global data)
- .stack and .sysstack sections (the primary and secondary system stacks)
- .sysmem section (dynamic memory space)
- .const section

There is no restriction on the size or placement of .text sections (code), .switch sections (switch statements), or .cinit/.pinit sections (variable initialization).

In the small model, the compiler uses 16-bit data pointers to access data. The upper 7 bits of the XAR*n* registers are set to point to the page that contains the .bss section. They remain set to that value throughout program execution.

6.1.2 Large Memory Model

The large memory model supports an unrestricted placement of data. To use the large memory model, use the –ml shell option.

In the large model:

- ❏ data pointers are 23 bits and occupy two words when stored in memory.
- ❏ the .stack and .sysstack sections must be on the same page.

Note:

Only code sections are allowed to cross page boundaries. Any other type of section must fit on one page.

When you compile code for the large memory model, you must link with the rts55x.lib runtime library. Note that you must use the same memory model for all of the files in your application. The linker will not accept a mix of large memory model and small memory model code.

6.1.3 Sections

The compiler produces relocatable blocks of code and data. These blocks are called *sections*. These sections are allocated into memory in a variety of ways to conform to a variety of system configurations. For more information about COFF sections, see the *Introduction to Common Object File Format* chapter in the *TMS320C55x Assembly Language Tools User's Guide*.

There are two basic types of sections:

- ❏ **Initialized sections** contain data or executable code. The C/C++ compiler creates the following initialized sections:
  - ■ The **.cinit section** contains tables for initializing variables and constants.
  - ■ The **.pinit section** contains the table for calling global object constructors at runtime.
  - ■ The **.const section** contains string constants and data defined with the C/C++ qualifier *const* (provided the constant is not also defined as *volatile*).
  - ■ The **.switch section** contains tables for switch statements.
  - ■ The **.text section** contains all the executable code.
- ❏ **Uninitialized sections** reserve space in memory (usually RAM). A program can use this space at runtime for creating and storing variables. The compiler creates the following uninitialized sections:

- The **.bss section** reserves space for global and static variables. At boot or load time, the C boot routine or the loader copies data out of the .cinit section (which may be in ROM) and uses it for initializing variables in .bss.
- The **.stack section** allocates memory for the system stack. This memory passes variables and is used for local storage.
- The **.sysstack section** allocates memory for the secondary system stack.

  Note that the .stack and .sysstack sections must be on the same page.
- The **.sysmem section** reserves space for dynamic memory allocation. This space is used by the malloc, calloc, and realloc functions. If a C/C++ program does not use these these functions, the compiler does not create the .sysmem section.
- The **.cio section** supports C I/O. This space is used as a buffer with the label __CIOBUF_. When any type of C I/O is performed (printf, scanf, etc.), the buffer is created. It contains an internal C I/O command (and required parameters) for the type of stream I/O that is to be performed, as well as the data being returned from the C I/O command. The .cio section must be allocated in the linker command file in order to use C I/O.

Only code sections are allowed to cross page boundaries. Any other type of section must fit on one page.

Note that the assembler creates an additional section called .data; the C/C++ compiler does not use this section.

The linker takes the individual sections from different modules and combines sections that have the same name. The resulting eight output sections and the appropriate placement in memory for each section are listed in Table 6–1. You can place these output sections anywhere in the address space, as needed to meet system requirements.

Table 6–1. Summary of Sections and Memory Placement

Section	Type of Memory	Section	Type of Memory
.bss	RAM	.data	ROM or RAM
.cinit	ROM or RAM	.text	ROM or RAM
.pinit	ROM or RAM	.stack	RAM
.cio	RAM	.sysstack	RAM
.const	ROM or RAM	.sysmem	RAM

For more information about allocating sections into memory, see the *Introduction to Common Object File Format* chapter, in the *TMS320C55x Assembly Language Tools User's Guide*.

6.1.4 C/C++ System Stack

The C/C++ compiler uses a stack to:

- ❏ Allocate local variables
- ❏ Pass arguments to functions
- ❏ Save the processor status

The runtime stack is allocated in a single continuous block of memory and grows down from high addresses to lower addresses. The compiler uses the hardware stack pointer (SP) to manage the stack.

The code doesn't check to see if the runtime stack overflows. Stack overflow occurs when the stack grows beyond the limits of the memory space that was allocated for it. Be sure to allocate adequate memory for the stack.

C55x also supports a secondary system stack. For compatibility with C54x, the primary runtime stack holds the lower 16 bits of addresses. The secondary system stack holds the upper 8 bits of C55x return addresses. The compiler uses the secondary stack pointer (SSP) to manage the secondary system stack.

The size of both stacks is set by the linker. The linker also creates global symbols, __STACK_SIZE and __SYSSTACK_SIZE, and assigns them a value equal to the respective sizes of the stacks in bytes. The default stack size is 1000 bytes. The default secondary system stack size is also 1000 bytes. You can change the size of either stack at link time by using the –stack or –sysstack options on the linker command line and specifying the size as a constant immediately after the option.

Note:

The .stack and .sysstack sections must be on the same page.

6.1.5 Dynamic Memory Allocation

The runtime-support library supplied with the compiler contains several functions (such as malloc, calloc, and realloc) that allow you to dynamically allocate memory for variables at runtime. Dynamic allocation is provided by standard runtime-support functions.

Memory is allocated from a global pool or heap that is defined in the .sysmem section. You can set the size of the .sysmem section by using the –heap *size* option with the linker command. The linker also creates a global symbol, __SYSMEM_SIZE, and assigns it as a value equal to the size of the heap in bytes. The default size is 2000 bytes. For more information on the –heap option, see Section 4.4, *Linker Options*, on page 4-6.

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

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

You can use a pointer and call the malloc function:

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

6.1.6 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 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 run, you can avoid having the .cinit section occupy space in memory. A loader can read the initialization tables directly from the object file (instead of from ROM) and perform the initialization directly at load time instead of at runtime. You can specify this to the linker by using the –cr linker option. For more information, see Section 6.8, *System Initialization*, on page 6-36.

6.1.7 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.

6.1.8 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.

When a structure contains a 32-bit (long) member, the long is aligned to a 2-word (32-bit) boundary. This may require padding before, inside, or at the end of the structure to ensure that the long is aligned accordingly and that the sizeof value for the structure is an even value.

All non-field 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 most significant bits of the structure word are filled first.

6.2 Character String Constants

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

- To initialize an array of characters. For example:

  ```
  char s[] = ″abc″;
  ```

  When a string is used as an initializer, it is simply treated as an initialized array; each character is a separate initializer. For more information about initialization, see Section 6.8, *System Initialization*, on page 6-36.

- 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. The following example defines 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. The number begins with 1 and is increased by 1 for each string defined. All strings used in a source module are defined at the end of the compiled assembly language module.

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

  If the same string is used more than once within a source module, the compiler attempts to minimize the number of definitions of the string by placing definitions in memory such that multiple uses of the string are in range of a single definition.

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

  ```
  char *a = ″abc″;
  a[1] = ′x′;        /* Incorrect! */
  ```

6.3 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 understand and follow these register conventions.

The register conventions dictate how the compiler uses registers and how values are preserved across function calls. Table 6–2 describes how the registers are used and how they are preserved. The parent function is the function making the function call. The child function is the function being called. For more information about how values are preserved across calls, see Section 6.4, *Function Structure and Calling Conventions*, on page 6-12.

Registers not used by compiled code are not listed in the table.

Table 6–2. Register Use and Preservation Conventions

Register	Preserved By	Uses
AC0, AC1, AC2, AC3	Parent	16-bit, 32-bit, or 40-bit data, or 24-bit code pointers
(X)AR0– (X)AR4	Parent	16-bit or 23-bit pointers, or 16-bit data
(X)AR5– (X)AR7	Child	
T0, T1	Parent	16-bit data
T2, T3	Child	
ST0_55, ST1_55, ST2_55, ST3_55	See Section 6.3.1	—
RPTC	Parent	—
CSR	Parent	—
BRC0, BRC1	Parent	—
BRS1	Parent	—
RSA0, RSA1	Parent	—
REA0, REA1	Parent	—
SP	N/A†	—
SSP	N/A	—
PC	N/A	—
RETA	Child	—
CFCT	Child	—

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

6.3.1 Status Registers

Table 6–3 shows the status register fields.

The Presumed Value column contains the value that:

- the compiler expects in that field upon entry to, or return from, a function.
- an assembly function can expect when the function is called from C/C++ code.
- an assembly function must set upon returning to, or making a call into, C/C++ code. If this is not possible, the assembly function cannot be used with C/C++ functions.

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.

The runtime initialization code (boot.asm) sets the assumed values. It also enables interrupts by clearing INTM in ST1_55. For more information on boot.asm, see Section 6.8, *System Initialization*, on page 6-36.

Table 6–3. Status Register Fields

(a) ST0_55

Field	Name	Presumed Value	Modified
ACOV[0-3]	Overflow detection	–	Yes
CARRY	Carry	–	Yes
TC[1-2]	Test control	–	Yes
DP[07-15]	Data page register	–	No

Table 6–3. Status Register Fields (Continued)

(b) ST1_55

Field	Name	Presumed Value	Modified
BRAF	Block-repeat active flag	–	No
CPL	Compiler mode	1	No
XF	External flag	–	No
HM	Hold mode	–	No
INTM	Interrupt mode	–	No
M40	Computation mode (D unit)	0	Yes†
SATD	Saturation mode (D unit)	0	Yes
SXMD	Sign-extension mode (D unit)	1	No
C16	Dual 16-bit arithmetic mode	0	No
FRCT	Fractional mode	0	Yes
54CM	C54x compatibility mode	0	Yes‡
ASM	Accumulator shift mode	–	No

† When 40-bit arithmetic is used (long long data type)
‡ When pragma C54X_CALL is used

(c) ST2_55

Field	Name	Presumed Value	Modified
ARMS	AR mode	1	No
DBGM	Debug enable mask	–	No
EALLOW	Emulation access enable	–	No
RDM	Rounding mode	0	No
CDPLC	CDP linear/circular configuration	0	No
AR[0-7]LC	AR[0-7] linear/circular configuration	0	No

(d) ST3_55

Field	Name	Presumed Value	Modified
CAFRZ	Cache freeze	–	No
CAEN	Cache enable	–	No
CACLR	Cache clear	–	No
HINT	Host interrupt	–	No
CBERR	CPU bus error flag	–	No
MPNMC	Microprocessor / microcomputer mode	–	No
SATA	Saturate mode (A unit)	0	Yes
CLKOFF	CLKOUT disable	–	No
SMUL	Saturation-on-multiplication mode	0	Yes
SST	Saturation-on-store mode	–	No

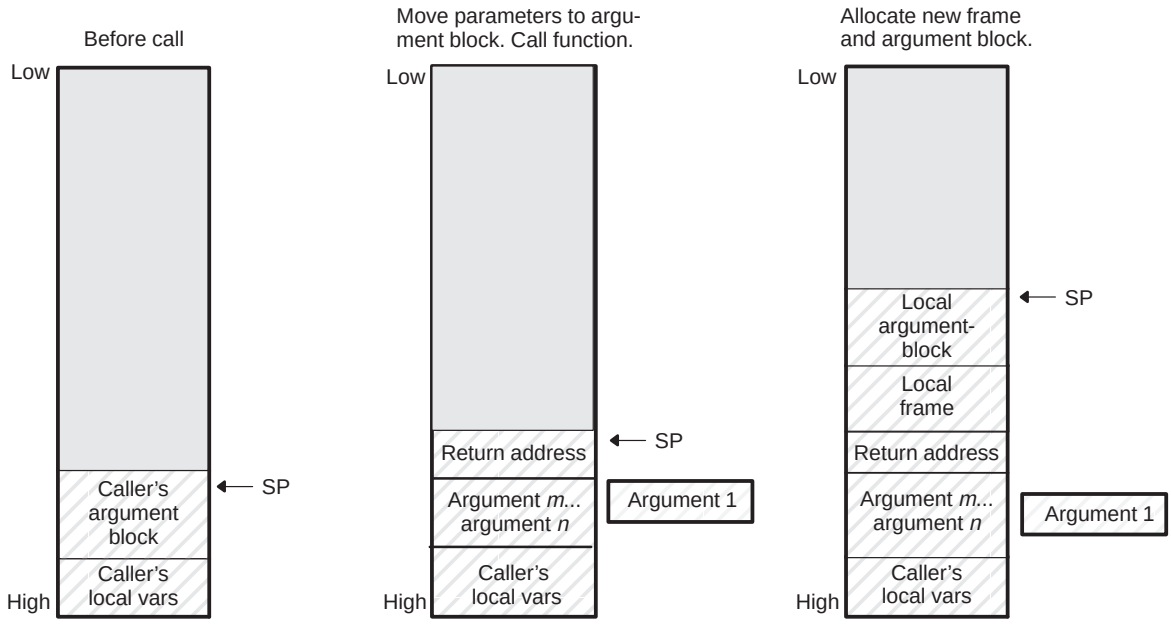
6.4 Function Structure and Calling Conventions

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

Figure 6–1 illustrates a typical function call. In this example, parameters are passed to the function, and the function uses local variables and calls another function. Note that up to 10 parameters are passed in registers. This example also shows allocation of a local frame and argument block for the called function. The stack must be aligned to a 32-bit boundary. The parent function pushes the 16-bit return PC.

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 than pushing them on the stack. The local frame and argument block are allocated at the same time.

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



6.4.1 How a Function Makes a Call

A function (parent function) performs the following tasks when it calls another function.

1) Arguments to a function are placed in registers or on the stack.

   a) For a function declared with an ellipsis (indicating that it is called with varying numbers of arguments), the last explicitly-declared argument is passed on the stack, followed by the rest of the arguments. Its stack address can act as a reference for accessing the undeclared arguments.

      Arguments declared before the last explicit argument follow rules shown below.

   b) In general, when an argument is passed to a function, the compiler assigns to it a particular class. It then places it in a register (if available) according to its class. The compiler uses three classes:

      - data pointer (int *, long *, etc.)
      - 16-bit data (char, short, int)
      - 32-bit data (long, float, double, function pointers)

      If the argument is a pointer to any type of data, it is considered a data pointer. If an argument will fit into a 16-bit register, it is considered 16-bit data. Otherwise, it is considered 32-bit data.

   c) A structure of two words (32 bits) or less is treated like a 32-bit data argument. It is passed in a register, if available.

   d) A structure larger than two words is passed by reference. The compiler will pass the address of the structure as a pointer. This pointer is treated like a data pointer argument.

   e) If the called (child) function returns a struct or union value, the parent function allocates space on the local stack for a structure of that size. The parent then passes the address of that space as a hidden first argument to the called function. This pointer is treated like a data pointer argument. In effect, the function call is transformed from:

      ```
      struct s result = fn(x, y);
      ```

      to

      ```
      fn(&result, x, y);
      ```

f) The arguments are assigned to registers in the order that the arguments are listed in the prototype. They are placed in the following registers according to their class, in the order shown below. For example, the first 32-bit data argument will be placed in AC0. The second 32-bit data argument will be placed in AC1, and so on.

Argument class	Assigned to register(s)
data pointer (16 or 23 bits)	(X)AR0, (X)AR1, (X)AR2, (X)AR3, (X)AR4
16-bit data	T0, T1, AR0, AR1, AR2, AR3, AR4
32-bit data	AC0, AC1, AC2

Note the overlap of AR*x* registers for data pointers and 16-bit data. If, for example, T0 and T1 hold 16-bit data arguments, and AR0 already holds a data pointer argument, a third 16-bit data argument would be placed in AR1. For an example, see the second prototype in Example 6–1.

If there are no available registers of the appropriate type, the argument goes on the stack.

g) Arguments passed on the stack are handled as follows. The stack is initially aligned to an even boundary. Then, each argument is aligned on the stack as appropriate for the argument's type (even alignment for long, long long, float, double, code pointers, and large model data pointers; no alignment for int, short, char, ioport pointers, and small model data pointers). Padding is inserted as necessary to bring arguments to the correct alignment.

2) The child function will save all of the save-on-entry registers (T2, T3, AR5–AR7). However, the parent function must save the values of the other registers, if they are needed after the call, by pushing the values onto the stack.

3) The parent calls the function.

4) The parent collects the return value.

- Short data values are returned in T0.
- Long data values are returned in AC0.
- Data pointer values are returned in (X)AR0.
- If the child returns a structure, the structure is on the local stack in the space allocated as described in step 1.

Examples of the register argument conventions are shown in Example 6–1. The registers in the examples are assigned according to the rules specified in steps 1 through 4 above.

Example 6–1. Register Argument Conventions

```
struct big { long x[10]; };
struct small { int x; };

T0   T0      AC0     AR0
int fn(int i1, long l2, int *p3);

AC0     AR0      T0     T1     AR1
long fn(int *p1, int i2, int i3, int i4);

AR0          AR1
struct big fn(int *p1);

T0     AR0          AR1
int fn(struct big b, int *p1);

AC0            AR0
struct small fn(int *p1);

T0     AC0            AR0
int fn(struct small b, int *p1);

T0       stack        stack...
int printf(char *fmt, ...);

         AC0      AC1      AC2      stack     T0
void fn(long l1, long l2, long l3, long l4, int i5);


         AC0      AC1      AC2     AR0      AR1
void fn(long l1, long l2, long l3, int *p4, int *p5,
        AR2      AR3     AR4      T0      T1
      int *p6, int *p7, int *p8, int  i9, int i10);
```

6.4.2 How a Called Function Responds

A called function performs the following tasks:

1) The called (child) function allocates enough space on the stack for any local variables, temporary storage areas, and arguments to functions that this function might call. This allocation occurs once at the beginning of the function.

2) If the child function modifies a save-on-entry register (T2, T3, AR5–AR7), it must save the value, either to the stack or to an unused register. The called function can modify any other register (ACx, Tx, ARx) without saving the value.

3) If the child 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. The local copy of the structure must be created from the passed pointer. If no writes are made to the structure, it can be referenced in the called function indirectly through the pointer argument.

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

5) If the child function returns a value, it places the value accordingly: long data values in AC0, short data values in T0, or data pointers in (X)AR0.

   If the child returns a structure, the parent allocates space for the structure and passes a pointer to this space in (X)AR0. To return the structure, the called function copies the structure to the memory block pointed to by the extra argument.

   If the parent does not use the return structure value, an address value of 0 can be passed in (X)AR0. This directs the child 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 the caller properly sets up the first argument) and where they are defined (so the function knows to copy the result).

6) The child function restores all registers saved in step 2.

7) The child function restores the stack to its original value.

8) The function returns.

6.4.3 Accessing Arguments and Locals

The compiler uses the compiler mode (selected when the CPL bit in status register ST1_55 is set to 1) for accessing arguments and locals. When this bit is set, the direct addressing mode computes the data address by adding the constant in the dma field of the instruction to the SP. For example:

```
MOV *SP(#8), T0
```

The largest offset available with this addressing mode is 128. So, if an object is too far away from the SP to use this mode of access, the compiler copies the SP to AR6 (FP) in the function prolog, then uses long offset addressing to access the data. For example:

```
MOV SP, FP
...
MOV *FP(#130), AR3
```

6.5 Interfacing C/C++ With Assembly Language

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

- ❏ Use separate modules of assembled code and link them with compiled C/C++ modules (see subsection 6.5.1). This is the most versatile method.
- ❏ Use assembly language variables and constants in C/C++ source (see subsection 6.5.2 on page 6-20).
- ❏ Use inline assembly language embedded directly in the C/C++ source (see subsection 6.5.3 on page 6-23).
- ❏ Use intrinsics in C/C++ source to directly call an assembly language statement (see subsection 6.5.4 on page 6-24).

6.5.1 Using Assembly Language Modules with C/C++ Code

Interfacing C/C++ with assembly language functions is straightforward if you follow the register conventions defined in Section 6.3, *Register Conventions*, and the calling conventions defined in Section 6.4, *Function Structure and Calling Conventions*. 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:

- ❏ You must preserve any dedicated registers modified by a function. Dedicated registers include:
  - ■ Save-on-entry registers (T2, T3, AR5, AR6, AR7)
  - ■ Stack pointer (SP)

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

  Any register that is not dedicated can be used freely without first being saved.
- ❏ To call a C54x assembly function from compiled C55x code, use the C54X_CALL or C54X_FAR_CALL pragma. For more information, see Section 5.7.2, *The C54X_CALL and C54X_FAR_CALL Pragmas*, on page 5-17.
- ❏ Interrupt routines must save *all* the registers they use. For more information, see Section 6.7, *Interrupt Handling*, on page 6-34.

- When calling C/C++ functions, remember that only the dedicated registers are preserved. C/C++ functions can change the contents of any other register.

- The compiler assumes that the stack is initialized at runtime to an even word address. If your assembly function calls a C/C++ function, you must align the SP by subtracting an odd number from the SP. The space must then be deallocated at the end of the function by adding the same number to the SP. For example:

```
_func: AADD #–1, SP    ; aligns SP
       ...             ; body of function
       AADD #1, SP     ; deallocate stack space
       RET             ; return from asm function
```

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

- Functions must return values as described in subsection 6.4.2, *How a Called Function Responds*, on page 6-15.

- 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 causes unpredictable results.

- The compiler places an underscore (_) at the beginning of all identifiers. This name space is reserved by the compiler. Prefix the names of variables and functions that are accessible from C/C++ with _. For example, a C/C++ variable called x is called _x in assembly language.

  For identifiers that are to be used only in an assembly language module or modules, the identifier should not begin with an underscore.

- Any object or function declared in assembly language that is to be accessed or called from C/C++ must be declared with the .global directive in the assembler. This defines the symbol as external and allows the linker to resolve references to it.

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

- Because compiled code runs with the CPL (compiler mode) bit set to 1, the only way to access directly addressed objects is with indirect absolute mode. For example:

```
MOV *(#global_var),AR3   ; works with CPL == 1
MOV global_var, AR3   ; does not work with CPL == 1
```

If you set the CPL bit to 0 in your assembly language function, you must set it back to 1 before returning to compiled code.

Example 6–2. Calling an Assembly Language Function From C

(a) C program

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

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

(b) Assembly language program

```
_asmfunc:

   ADD *AR0, T0, T0     ; add gvar to T0 => i is in T0
   RETURN              ; start return
```

In the assembly language code in Example 6–2, note the underscore on the C/C++ symbol name used in the assembly code.

The parameter i is passed in T0. Also note the use of indirect absolute mode to access gvar. Because the CPL bit is set to 1, direct addressing mode adds the dma field to the SP. Thus, direct addressing mode cannot be used to access globals.

6.5.2 Accessing Assembly Language Variables From C/C++

It is sometimes useful for a C/C++ program to access variables defined in assembly language. There are three methods that you can use to accomplish this, depending on where and how the item is defined: a variable defined in the .bss section, a variable not defined in the .bss section, or a constant.

6.5.2.1 Accessing Assembly Language Global Variables

Accessing uninitialized variables from the .bss section or a section named with .usect is straightforward:

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

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

Example 6–3. Accessing a Variable From C

(a) Assembly language program

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

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

(b) C program

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

You may not always want a variable to be in the .bss section. For example, a common situation is a lookup table defined in assembly language that you don't want to put in RAM. In this case, you must define a pointer to the object and access it indirectly from C/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/C++, you must declare the object as *extern* and not precede it with an underscore. Then you can access the object normally.

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

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

(a) C Program

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

(b) Assembly Language Program

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

6.5.2.2 Accessing Assembly Language Constants

You can define global constants in assembly language by using the .set and .global directives, or you can define them in a linker command file using a 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 6–5.

Example 6–5. Accessing an Assembly Language Constant From C

(a) Assembly language program

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

(b) 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 */
```

Since you are referencing only the symbol's value as stored in the symbol table, the symbol's declared type is unimportant. In Example 6–5, int is used. You can reference linker-defined symbols in a similar manner.

6.5.3 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 created by the compiler. A series of asm statements places sequential lines of assembly language into the compiler output with no intervening code. For more information, see Section 5.6, *The asm Statement*, on page 5-15.

The asm statement is useful for inserting comments in the compiler output. Simply start the assembly code string with a semicolon (;) as shown below:

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

Note: Using the asm Statement

Keep the following in mind when using 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 can produce unpredictable results by confusing the register-tracking algorithms that the code generator uses.
- Do not change the value of a C/C++ variable when using an asm statement.
- Do not use the asm statement to insert assembler directives that change the assembly environment.

6.5.4 Using Intrinsics to Access Assembly Language Statements

The 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 a leading underscore, and are accessed by calling them as you do a function. For example:

```
int x1, x2, y;
y = _sadd(x1, x2);
```

The intrinsics listed in Table 6–4 are included. Use gsm.h, the header file shown in Figure 6–2 on page 6-28, to map certain intrinsics onto European Telecommunications Standards Institute (ETSI) functions. Additional support for ETSI functions is described in Section 6.5.4.1 on page 6-27.

The compiler also supports an “associative” version for some of the saturating intrinsics. These associative intrinsics are prefixed with “_a_”. When these intrinsics are used, the compiler is able to reorder their arithmetic computations, which may produce more efficient code. However, the reordering may also result in saturation at different points in the computation if (and only if) computation overflows at some point. This difference may affect the result; consequently, a bit-exact comparison with a non-associative intrinsic could fail.

Table 6–4. TMS320C55x C/C++ Compiler Intrinsics

Compiler Intrinsic	Description
short _abs(short src);	Creates a 16-bit absolute value.
long _labs(long src);	Creates a 32-bit absolute value.
short _abss(short src);	Creates a saturated 16-bit absolute value. _abss(0x8000) => 0x7FFF (SATA set)
long _labss(long src);	Creates a saturated 32-bit absolute value. _labss(0x8000000) => 0x7FFFFFFF (SATD set)
long long _llabss(long long src)	Creates a saturated 40-bit absolute value. (SATD set)
short _norm(short src);	Produces the number of left shifts needed to normalize src.
short _lnorm(long src);	Produces the number of left shifts needed to normalize src.

Table 6–4. TMS320C55x C/C++ Compiler Intrinsics (Continued)

Compiler Intrinsic	Description
long _rnd(long src);	Rounds src by adding 2^{15} and then zeroing out the lower 16 bits. Produces a 32-bit saturated result. (SATD set)
short _sadd(short src1, short src2); short _a_sadd(short src1, short src2);	Adds two 16-bit integers, with SATA set, producing a saturated 16-bit result.
long _lsadd(long src1, long src2); long _a_lsadd(long src1, long src2);	Adds two 32-bit integers, with SATD set, producing a saturated 32-bit result.
long long _llsadd(long long src1, long long src2); long long _a_llsadd(long long src1, long long src2);	Adds two 40-bit integers, with SATD set, producing a saturated 40-bit result.
long _smac(long src, short op1, short op2); long _a_smac(long src, short op1, short op2);	Multiplies op1 and op2, shifts the result left by 1, and adds it to src. Produces a saturated 32-bit result. (SATD , SMUL, and FRCT set)
long _smacr(long src, short op1, short op2); long _a_smacr(long src, short op1, short op2);	Multiplies op1 and op2, shifts the result left by 1, adds the result to src, and then rounds the result by adding 2^{15} and zeroing out the lower 16 bits. (SATD , SMUL, and FRCT set)
long _smas(long src, short op1, short op2); long _a_smas(long src, short op1, short op2);	Multiplies op1 and op2, shifts the result left by 1, and subtracts it from src. Produces a 32-bit result. (SATD, SMUL, and FRCT set)
long _smasr(long src, short op1, short op2); long _a_smasr(long src, short op1, short op2);	Multiplies op1 and op2, shifts the result left by 1, subtracts the result from src, and then rounds the result by adding 2^{15} and zeroing out the lower 16 bits. (SATD , SMUL, and FRCT set)
short _smpy(short src1, short src2);	Multiplies src1 and src2, and shifts the result left by 1. Produces a saturated 16-bit result. (SATD and FRCT set)
long _lsmpy(short src1, short src2);	Multiplies src1 and src2, and shifts the result left by 1. Produces a saturated 32-bit result. (SATD and FRCT set)
long _smpyr(short src1, short src2);	Multiplies src1 and src2, shifts the result left by 1, and rounds by adding 2^{15} to the result and zeroing out the lower 16 bits. (SATD and FRCT set)

Table 6–4. TMS320C55x C/C++ Compiler Intrinsics (Continued)

Compiler Intrinsic	Description
short _sneg(short src);	Negates the 16-bit value with saturation. _sneg(0xffff8000) => 0x00007FFF
long _lsneg(long src);	Negates the 32-bit value with saturation. _lsneg(0x80000000) => 0x7FFFFFFF
long long _llsneg(long long src);	Negates the 40-bit value with saturation.
short _sshl(short src1, short src2);	Shifts src1 left by src2 and produces a 16-bit result. (SATD set)
long _lsshl(long src1, short src2);	Shifts src1 left by src2 and produces a 32-bit result. (SATD set)
short _ssub(short src1, short src2);	Subtracts src2 from src1 with SATA set, producing a saturated 16-bit result.
long _lssub(long src1, long src2);	Subtracts src2 from src1 with SATD set, producing a saturated 32-bit result.
long long _llssub(long long src1, long long src2)	Subtracts src2 from src1 with SATD set, producing a saturated 40-bit result.
long _lsat(long long src);	Converts a 40-bit long long to a 32-bit long and saturates if necessary.
short _shrs(short src1, short src2);	Shifts src1 right by src2 and produces a 16-bit result. (SATD set)
long _lshrs(long src1, short src2);	Shifts src1 right by src2 and produces a 32-bit result. (SATD set)

6.5.4.1 Intrinsics and ETSI functions

The functions in Table 6–5 provide additional ETSI support for the intrinsics functions. Functions L_add_c, L_sub_c, and L_sat map to ETSI inline macros. The other functions in the table are runtime functions.

Table 6–5. ETSI Support Functions

Compiler Intrinsic	Description
long L_add_c(long src1, long src2);	Adds src1, src2, and Carry bit. This function does not map to a single assembly instruction, but to an inline function.
long L_sub_c(long src1, long src2);	Subtracts src2 and logical inverse of sign bit from src1. This function does not map to a single assembly instruction, but to an inline function.
long L_sat(long src1);	Saturates any result after L_add_c or L_sub_c if Overflow is set.
int crshft_r(int x, int y);	Shifts x right by y, rounding the result with saturation.
long L_crshft_r(long x, int y);	Shifts x right by y, rounding the result with saturation.
int divs(int x, int y);	Divides x by y with saturation.

Figure 6–2. Intrinsics Header File, gsm.h

```
#ifndef _GSMHDR
#define _GSMHDR
#include <linkage.h>
#define MAX_16 0x7fff
#define MIN_16 –32768
#define MAX_32 0x7fffffff
#define MIN_32 0x80000000

extern int Overflow;
extern int Carry;

#define L_add(a,b) (_lsadd((a),(b)))
#define L_sub(a,b) (_lssub((a),(b)))
#define L_negate(a) (_lsneg(a))
#define L_deposit_h(a) ((long)a<<16)
#define L_deposit_l(a) ((long)a)
#define L_abs(a) (_labss((a)))
#define L_mult(a,b) (_lsmpy((a),(b)))
#define L_mac(a,b,c) (_smac((a),(b),(c)))
#define L_macNs(a,b,c) (L_add_c((a),L_mult((b),(c))))
#define L_msu(a,b,c) (_smas((a),(b),(c)))
#define L_msuNs(a,b,c) (L_sub_c((a),L_mult((b),(c))))
#define L_shl(a,b) _lsshl((a),(b))
#define L_shr(a,b) _lshrs((a),(b))
#define L_shr_r(a,b) (L_crshft_r((a),(b)))
#define L_shift_r(a,b) (L_shr_r((a),–(b)))


#define abs_s(a) (_abss((a)))
#define add(a,b) (_sadd((a),(b)))
#define sub(a,b) (_ssub((a),(b)))
#define extract_h(a) ((unsigned)((a)>>16))
#define extract_l(a) ((int)a)
#define round(a) (short)(_rnd(a)>>16)
#define mac_r(a,b,c) (short)(_smacr((a),(b),(c))>>16)
#define msu_r(a,b,c) (short)(_smasr((a),(b),(c))>>16)
#define mult_r(a,b) (short)(_smpyr((a),(b))>>16)
#define mult(a,b) (_smpy((a),(b)))
#define norm_l(a) (_lnorm(a))
#define norm_s(a) (_norm(a))
#define negate(a) (_sneg(a))
#define shl(a,b) _sshl((a),(b))
#define shr(a,b) _shrs((a),(b))
#define shr_r(a,b) (crshft_r((a),(b)))
#define shift_r(a,b) (shr_r(a,–(b)))
#define div_s(a,b) (divs(a,b))
```

Figure 6–2. Intrinsics Header File, gsm.h (Continued)

```
#ifdef __cplusplus
extern "C"
{
#endif /* __cplusplus */

int          crshft_r(int x, int y);
long         L_crshft_r(long x, int y);
int          divs(int x, int y);
_IDECL long  L_add_c(long, long);
_IDECL long  L_sub_c(long, long);
_IDECL long  L_sat(long);

#ifdef _INLINE
static inline long L_add_c (long L_var1, long L_var2)
{
   unsigned  long   uv1 = L_var1;
   unsigned  long   uv2 = L_var2;
   int       cin = Carry;
   unsigned  long   result =  uv1 + uv2 + cin;

   Carry = ((~result & (uv1 | uv2)) | (uv1 & uv2)) >> 31;
   Overflow = ((~(uv1 ^ uv2)) & (uv1 ^ result)) >> 31;

   if (cin && result == 0x80000000) Overflow = 1;
   return (long)result;
}

static inline long L_sub_c (long L_var1, long L_var2)
{
   unsigned  long   uv1 =  L_var1;
   unsigned  long   uv2 =  L_var2;
   int       cin = Carry;
   unsigned  long   result =  uv1 + ~uv2 + cin;

   Carry  =  ((~result & (uv1 | ~uv2)) | (uv1 & ~uv2)) >> 31;
   Overflow  =  ((uv1 ^ uv2) & (uv1 ^ result)) >> 31;

   if (!cin && result == 0x7fffffff) Overflow = 1;
   return (long)result;
}

static inline long L_sat (long L_var1)
{
   int cin   =  Carry;
   return !Overflow ? L_var1 : (Carry = Overflow = 0, 0x7fffffff+cin);
}
#endif /* !_INLINE */

#ifdef __cplusplus
} /* extern "C" */
#endif /* __cplusplus */
#endif /* !_GSMHDR */
```

6.6 DSP Arithmetic Operations

Table 6–6 lists some common DSP operations and how the compiler supports each operation. The operations are supported with either a C expression or an intrinsic. Intrinsics are explained in detail in Section 6.5.4, *Using Intrinsics to Access Assembly Language Statements*, on page 6-24.

Table 6–6. DSP Arithmetic Operations Supported by Compiler

(a) Multiply operations

Operation	C expression or intrinsic	Assembly generated by compiler
16 * 16 => 32	short a, b; long c; c = (long)a * b;	dst = src1 * src2
Q15 * Q15 => Q31 (fractional)	short a, b; long c; c = ((long)a*b) << 1;	dst = src1 * src2 (FRCT set)
Q15 * Q15 => Q15 (fractional with saturation)	short a, b; long c; c = _smpy(a,b);	tempAC = src1 * src2 dst = HI(tempAC) (FRCT, SATD set)
Q15 * Q15 => Q31 (fractional with saturation)	short a, b; long c; c = _lsmpy(a,b);	dst = src1 * src2 (FRCT, SATD set)

(b) Multiply-accumulate operations

Operation	C expression or intrinsic	Assembly generated by compiler
32 + 16 * 16 => 32	short a, b; long c; c = c + ((long)a * b);	dst = dst + (src1 * src2)
Q31 + Q15 * Q15 => Q31 (fractional)	short a, b; long c; c = c + ((long)a*b) << 1;	dst = dst + (src1 * src2) (FRCT set)
Q31 + Q15 * Q15 => Q31 (fractional with saturation)	short a, b; long c; c = _smac(c,a,b);	dst = dst + (src1 * src2) (FRCT, SATD, SMUL set)

Table 6–6. DSP Arithmetic Operations Supported by Compiler (Continued)

(c) Multiply-subtract operations

Operation	C expression or intrinsic	Assembly generated by compiler
32 – 16 * 16 => 32	short a, b; long c; c = c – ((long)a * b);	dst = dst – (src1 * src2)
Q31 – Q15 * Q15 => Q31 (fractional)	short a, b; long c; c = c – ((long)a*b) << 1;	dst = dst – (src1 * src2) (FRCT set)
Q31 – Q15 * Q15 => Q31 (fractional with saturation)	short a, b; long c; c = _smas(c,a,b);	dst = dst – (src1 * src2) (FRCT, SATD, SMUL set)

(d) Add operations

Operation	C expression or intrinsic	Assembly generated by compiler
16 + 16 => 16 32 + 32 => 32 40 + 32 => 40 40 + 40 => 40	*type* a, b, c; c = a + b;	dst = src1 dst = dst + src2
16 + 16 => 16 (with saturation)	short a, b, c; c = _sadd(a,b);	dst = src1 dst = dst + src2 (SATA set)
32 + 32 => 32 (with saturation)	long a, b, c; c = _lsadd(a,b);	dst = src1 dst = dst + src2 (SATD set)
40 + 40 => 40 (with saturation)	long long a, b, c; c = _llsadd(a,b);	dst = src1 dst = dst + src2 (SATD set)

Table 6–6. DSP Arithmetic Operations Supported by Compiler (Continued)

(e) Subtract operations

Operation	C expression or intrinsic	Assembly generated by compiler
16 – 16 => 16 32 – 32 => 32 40 – 32 => 40 40 – 40 => 40	*type* a, b, c; c = a – b;	dst = src1 dst = dst – src2
16 – 16 => 16 (with saturation)	short a, b, c; c = _ssub(a,b);	dst = src1 dst = dst – src2 (SATA set)
32 – 32 => 32 (with saturation)	long a, b, c; c = _lssub(a,b);	dst = src1 dst = dst – src2 (SATD set)
40 – 40 => 40 (with saturation)	long long a, b, c; c = _llssub(a,b);	dst = src1 dst = dst – src2 (SATD set)

(f) Absolute value operations

Operation	C expression or intrinsic	Assembly generated by compiler
|16| => 16 |32| => 32 |40| => 40	*type* a, b; b = a >= 0?a:–a;	dst = |src|
|16| => 16 (with saturation)	short a, b; b = _abss(a);	dst = |src| (SATA set)
|32| => 32 (with saturation)	long a, b; b = _labss(a);	dst = |src| (SATD set)
|40| => 40 (with saturation)	long long a, b; b = _llabss(a);	dst = |src| (SATD set)

(g) Size change operations

Operation	C expression or intrinsic	Assembly generated by compiler
40 => 32 (both Q31)	long long a; long b; b = a;	dst = src
40 => 32 (both Q31) (with saturation)	long long a; long b; b = _lsat(a);	dst = saturate(src)
32 => 16 (Q31 => Q15)	long a; short b; b = a >> 16;	dst = HI(src)

Table 6–6. DSP Arithmetic Operations Supported by Compiler (Continued)

(h) Format change operations †

Operation	C expression or intrinsic	Assembly generated by compiler
Q39 => Q31	long long a; long b; b = a >> 8;	dst = src >> 8
Q15 => Q12	short a; short b; b = a >> 3;	dst = src >> 3
Q30 => Q31 (with saturation)	long a; long b; b = _lsshl(a,1);	dst = src << 1 (SATD set)

† Only representative examples are shown in the table. Other format change operations are possible with shift operators or intrinsics.

(i) Rounding operations

Operation	C expression or intrinsic	Assembly generated by compiler
round(32) => 16 (toward infinity)	long a; short b; b = rnd(a);	dst = HI(rnd(src))
round(32) => 16 (with saturation)	long a; short b; b = _srnd(a);	dst = HI(saturate(rnd(src)))
round(32) => 16 (nearest	long a; short b; b = _rndn(a);	dst = HI(rnd(src)) (RDM set)
round(32) => 16 (nearest, with saturation)	long a; short b; b = _srndn(a);	dst = HI(saturate(rnd(src))) (RDM set)

6.7 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.

6.7.1 General Points About Interrupts

An interrupt routine may 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:

- ❏ It is your responsibility to handle any special masking of interrupts (via the IER0 register). You can use inline assembly to enable or disable the interrupts and modify the IER0 register without corrupting the C/C++ environment or C/C++ pointer.
- ❏ An interrupt handling routine cannot have arguments. If any are declared, they are ignored.
- ❏ An interrupt handling routine cannot be called by normal C/C++ code.
- ❏ 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 c_int00, you cannot assume that the runtime stack is set up; therefore, you *cannot allocate local variables*, and you *cannot save any information on the runtime stack*.
- ❏ To associate an interrupt routine with an interrupt, a branch must be placed in the appropriate interrupt vector. You can use the assembler and linker to do this by creating a simple table of branch instructions 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.
- ❏ Align the stack to an even (long-aligned) address.

6.7.2 Using C/C++ Interrupt Routines

Interrupts can be handled *directly* with C/C++ functions by using the interrupt keyword. For example:

```
interrupt void isr()
{
  ...
}
```

Adding the interrupt keyword defines an interrupt routine. When the compiler encounters one of these routines, it generates code that allows the function to be activated from an interrupt trap. This method provides more functionality than the standard C/C++ signal mechanism. This does not prevent implementation of the signal function, but it does allow these functions to be written entirely in C/C++.

6.7.3 Saving Context on Interrupt Entry

All registers that the interrupt routine uses, including the status registers, must be preserved. If the interrupt routine calls other functions, *all* of the registers in Table 6–2 on page 6-9 must be preserved.

6.8 System Initialization

Before you can run a C/C++ program, the C/C++ runtime environment must be created. This task is performed by the C/C++ boot routine, which is a function called _c_int00. The runtime-support source library (rts.src) contains the source to this routine in a module called boot.asm.

To begin running the system, the _c_int00 function can be called by reset hardware. You must link the _c_int00 function with the other object modules. This occurs automatically when you use the –c or –cr linker function option and include rts.src as one of the linker input files.

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 to initialize the C/C++ environment:

1) Sets up the stack and the secondary system stack.

2) Initializes global variables by copying the data from the initialization tables in the .cinit and .pinit sections to the storage allocated for the variables in the .bss section. If initializing variables at load time (–cr option), a loader performs this step before the program runs (it is not performed by the boot routine). For information, see subsection 6.8.1, *Automatic Initialization of Variables*.

3) Calls the function main to begin running the C/C++ program.

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

6.8.1 Automatic Initialization of Variables

Any global variables declared as preinitialized 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 that contain data for initializing global and static variables in a .cinit section in each file. Each compiled module contains these initialization tables. The linker combines them into a single table (a single .cinit section). The boot routine or 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 must be set to 0 before program execution. The C/C++ compiler does not perform any preinitialization of uninitialized variables. You must explicitly initialize any variable that must have an initial value of 0.

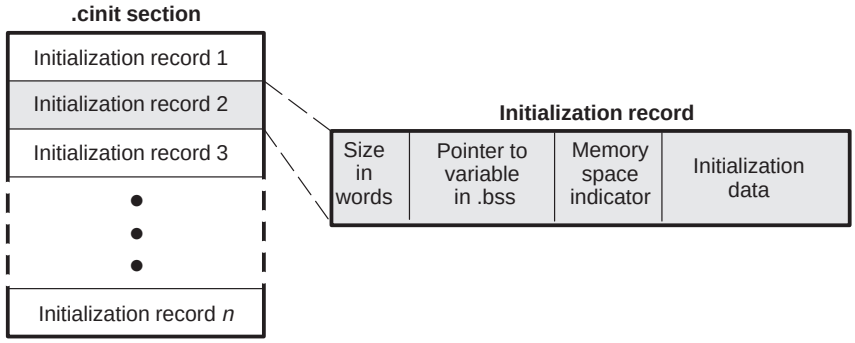
6.8.2 Global Constructors

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.

6.8.3 Initialization Tables

The tables in the .cinit section consist of variable-size initialization records. Each variable that must be autoinitialized has a record in the .cinit section. Figure 6–3 shows the format of the .cinit section and the initialization records.

Figure 6–3. Format of Initialization Records in the .cinit Section



An initialization record contains the following information:

- The first field (word 0) contains the size in words of the initialization data.

  Bits 14 and 15 are reserved. An initialization record can be up to 2^{13} words in length.

- The second field contains the starting address of the area in the .bss section where the initialization data must be copied. This field is 24 bits to accommodate an address greater than 16 bits.

- The third field contains 8 bits of flags. Bit 0 is a flag for the memory space indicator (I/O or data). The other bits are reserved.

- The fourth field (words 3 through *n*) contains the data that is copied to initialize the variable.

The .cinit section contains an initialization record for each variable that is initialized. Example 6–6 (a) shows initialized variables defined in C/C++. Example 6–6 (b) shows the corresponding initialization table.

Example 6–6. Initialization Variables and Initialization Table

(a) Initialized variables defined in C

```
int    i = 3;
long   x = 4;
float  f = 1.0;
char   s[] = “abcd”;
long   a[5] = { 1, 2, 3, 4, 5 };
```

Example 6–6. Initialization Variables and Initialization Table (Continued)

(b) Initialized information for variables defined in (a)

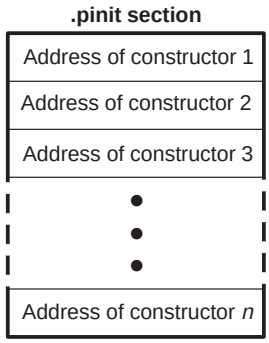
```
       .sect     ".cinit"      ; Initialization section
* Initialization record for variable i
       .field    1,16          ; length of data (1 word)
       .field    _i+0,24       ; address in .bss
       .field    0,8           ; signifies data memory
       .field    3,16          ; int is 16 bits

* Initialization record for variable x
       .field    2,16          ; length of data (2 words)
       .field    _l+0,24       ; address in .bss
       .field    0,8           ; data memory
       .field    4,32          ; long is 32 bits

* Initialization record for variable f
       .field    2,16          ; length of data (2 words)
       .field    _f+0,24       ; address in .bss
       .field    0,8           ; data memory
       .xlong    0x3f800000    ; float is 32 bits

* Initialization record for variable s
       .field    IR_1,16       ; length of data
       .field    _s+0,24       ; address in .bss
       .field    0,8           ; data memory
       .field    97,16         ; a
       .field    98,16         ; b
       .field    99,16         ; c
       .field    100,16        ; d
       .field    0,16          ; end of string
IR_1   .set      5             ; symbolic value gives
                               ; count of elements

* Initialization record for variable a
       .field    IR_2,16       ; length of data
       .field    _a+0,24       ; address in .bss
       .field    0,8           ; data memory
       .field    1,32          ; beginning of array
       .field    2,32
       .field    3,32
       .field    4,32
       .field    5,32          ; end of array
IR_2   .set      10            ; size of array
```

The .cinit section must contain only initialization tables in this format. If you interface assembly language modules to your C/C++ programs, do not use the .cinit section for any other purpose.

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

marks the end of the initialization tables.

Figure 6–4. Format of Initialization Records in the .pinit Section



Likewise, the –c or –cr linker option causes the linker to combine all of the .pinit sections from all the C/C++ modules and appends a null word to the end of the composite .pinit section. The boot routine knows the end of the global constructor table when it encounters a null constructor address.

Note that const-qualified variables are initialized differently; see subsection 5.9.1, *Initializing Static and Global Variables with the Const Type Qualifier*, on page 5-30.

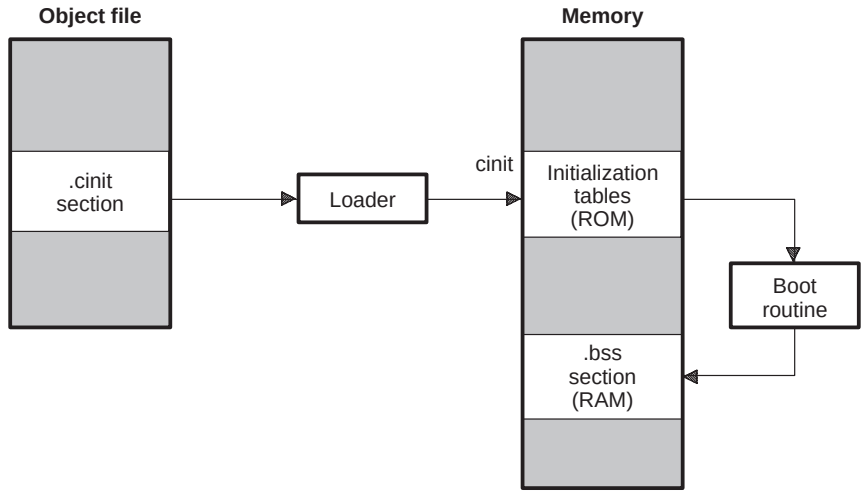
6.8.4 Autoinitialization of Variables at Runtime

Autoinitializing variables at runtime is the default model for autoinitialization. To use this method, invoke the linker with the –c option.

Using this method, the .cinit section is loaded into memory (possibly ROM) along with all the other initialized sections, and global variables are initialized at runtime. 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 6–5 illustrates autoinitialization at runtime. Use this method in any system where your application runs from code burned into ROM.

Figure 6–5. Autoinitialization at Runtime



6.8.5 Autoinitialization of Variables at Load Time

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

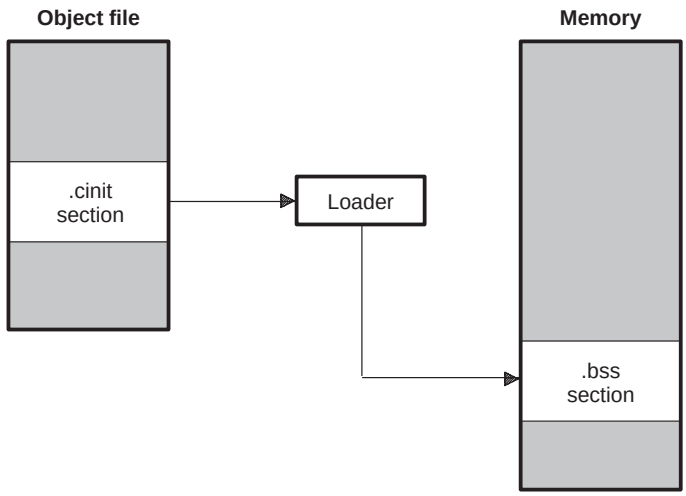
When you use the –cr linker option, the linker sets the STYP_COPY bit in the .cinit section's header. This tells the loader not to load the .cinit section into memory. (The .cinit section occupies no space in the memory map.) The linker also sets the cinit symbol to –1 (normally, cinit points to the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

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

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

Figure 6–6 illustrates the RAM model of autoinitialization.

Figure 6–6. Autoinitialization at Load Time



Chapter 7

Runtime-Support Functions

Some of the tasks that a C/C++ program performs (such as I/O, dynamic memory allocation, string operations, and string searches) are not part of the C/C++ language itself. The runtime-support functions, which are included with the C/C++ compiler, are standard ANSI functions that perform these tasks.

The runtime-support library, rts.src, contains the source for these functions as well as for other functions and routines. All of the ANSI functions except those that require an underlying operating system (such as signals) are provided.

A library-build utility is included with the code generation tools that lets you create customized runtime-support libraries. For information about using the library-build utility, see Chapter 8, *Library-Build Utility*.

Topic **Page**

7.1 Libraries

The following libraries are included with the TMS320C55x C/C++ compiler:

- *rts55.lib* contains the ANSI runtime-support object library
- *rts55x.lib* contains the ANSI runtime-support object library for the large memory model
- *rts.src* contains the source for the ANSI runtime-support routines

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

When you link your program, you must specify an object library as one of the linker input files so that references to the I/O and runtime-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 issues.

When a library is linked, the linker includes only those library members required to resolve undefined references. For more information about linking, see the linker description chapter of the *TMS320C55x Assembly Language Tools User's Guide*.

7.1.1 Modifying a Library Function

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

```
ar55 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(s) into the library:

```
cl55 –options atoi.c strcpy.c      ;recompile
ar55 –r rts.src atoi.c strcpy.c   ;rebuild library
```

You can also build a new library this way, rather than rebuilding into rts55.lib. For more information about the archiver, see the archiver description chapter of the *TMS320C55x Assembly Language Tools User's Guide*.

7.1.2 Building a Library With Different Options

You can create a new library from rts.src by using the library-build utility, mk55. For example, use this command to build an optimized runtime-support library:

```
mk55 ––u –o2 rts.src –l rts55.lib
```

The ––u option tells the mk55 utility to use the header files in the current directory, rather than extracting them from the source archive. The use of the optimizer (–o2) option does not affect compatibility with code compiled without this option. For more information about building libraries, see Chapter 8, *Library-Build Utility*.

7.2 The C I/O Functions

The C I/O functions make it possible to access the host's operating system to perform I/O (using the debugger). For example, printf statements executed in a C55x 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, or cstdio for C++ code, for each module that references a function.

If you do not use TI's default linker command file, you should allocate the .cio section in your linker command file. The .cio section provides a buffer (with the label __CIOBUF_) that is used by the runtimes and the debugger. When any type of C I/O is performed (printf, scanf, etc.), the buffer is created and placed at the __CIOBUF_ address. The buffer contains:

- An internal C I/O command (and required parameters) for the type of stream I/O that is to be performed.
- The data being returned from the I/O command.

The buffer will be read by the debugger, and the debugger will perform the appropriate I/O command.

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

```
#include <stdio.h>

main()
{
   FILE *fid;

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

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

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

```
cl55 main.c –z –heap 400 –l rts.lib –o main.out
```

Executing main.out under the C55x debugger on a SPARC host accomplishes the following:

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

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

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

7.2.1 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.

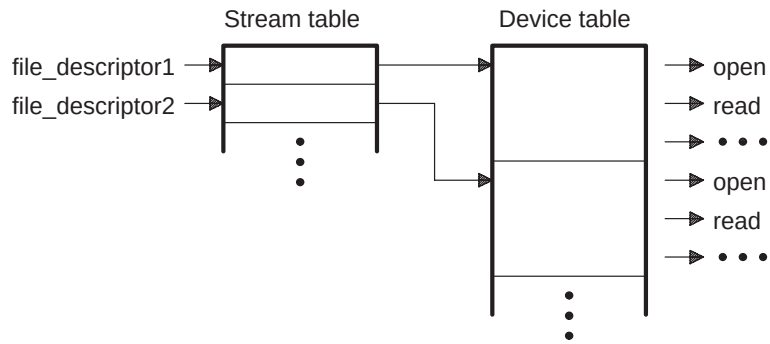
The high-level functions are the standard C library of stream I/O routines (printf, scanf, fopen, getchar, etc.). These routines map an I/O request to one or more of the I/O commands that are handled by the low-level shell.

The low-level functions are comprised 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.

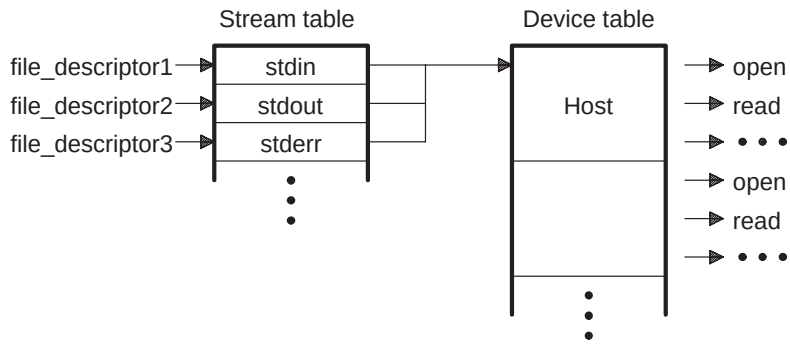
The data structures interact as shown in Figure 7–1.

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



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

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



At the next level are the user-definable device-level drivers. They map directly to the low-level I/O functions. The 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 on pages 7-9 through 7-12. 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.

7.2.2 Adding a Device For C I/O

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

1) Define the device-level functions as described in subsection 7.2.1 on page 7-6.

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 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 7-35.

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

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

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

main()
{
   FILE *fid;

   add_device("mydevice", _MSA, my_open, my_close, my_read, my_write, my_lseek,
                                my_unlink, my_rename);

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

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

   fclose(fid);
}
```

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.h>
#include <file.h>
int std::close(int *file_descriptor***);**

Description

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

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

Return Value

The return value is one of the following:

0	if successful
–1	if fails

lseek — *Set File Position Indicator*

Syntax for C

#include<stdio.h>
#include <file.h>
long lseek(int *file_descriptor***, long** *offset***, int** *origin***);**

Syntax for C++

#include<cstdio.h>
#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 return function is one of the following:

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

open

Open File or Device For I/O

Syntax for C

#include<stdio.h>
#include <file.h>
int open(const char ****path*, unsigned** *flags***, int** *mode***);**

Syntax for C++

#include<cstdio.h>
#include <file.h>
int std::open(const char ****path*, unsigned** *flags***, int** *mode***);**

Description

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

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

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

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

- The *mode* is required by ignored.

Return Value

The function returns one of the following values:

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

read — *Read Characters From Buffer*

Syntax for C

#include<stdio.h>
#include <file.h>
int read(int *file_descriptor*, **char** **buffer*, **unsigned** *count*);

Syntax for C++

#include<cstdio.h>
#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(const char ***_old_name_, const char ***_new_name_**);**

Syntax for C++

#include<cstdio.h>
#include <file.h>
int std::rename(const char ***_old_name_, const 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(const char *_path_);

Syntax for C++

#include<cstdio.h>
#include <file.h>
int std::unlink(const 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*, **const char *** *buffer*, **unsigned** *count*);

Syntax for C++

#include<cstdio.h>
#include <file.h>
int std::write(int *file_descriptor*, **const char *** *buffer*, **unsigned** *count*);

Description

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

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

Return Value

The function returns one of the following values:

#	number of characters written if successful
–1	if fails

7.3 Header Files

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

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

These are the header files that declare the ANSI C runtime-support functions:

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

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

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

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

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

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

Subsections 7.3.1 through 7.3.7 describe the header files that are included with the C/C++ compiler.

7.3.1 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 run time expression.

- If the expression is true (nonzero), the program continues running.
- If the expression is false, the macro outputs a message that contains the expression, the source file name, and the line number of the statement that contains the expression; then, the program terminates (using the abort function).

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

The assert function is listed in Table 7–3 (a) on page 7-25.

7.3.2 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*).

The character typing and conversion functions are listed in Table 7–3 (b) on page 7-25.

7.3.3 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)
- ENOENT for path errors (path does not exist)
- EFPOS for seek errors (file position error)

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

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

The file.h header declares the low-level I/O functions used to implement input and output operations. Section 7.2, *The C I/O Functions*, describes how to implement I/O for C55x.

7.3.5 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 processor's numeric representations. Table 7–1 and Table 7–2 list these macros and their associated limits.

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

Macro	Value	Description
CHAR_BIT	16	Number of bits in type char
SCHAR_MIN	–32 768	Minimum value for a signed char
SCHAR_MAX	32 767	Maximum value for a signed char
UCHAR_MAX	65 535	Maximum value for an unsigned char
CHAR_MIN	–32 768	Minimum value for a char
CHAR_MAX	32 767	Maximum value for a char
SHRT_MIN	–32 768	Minimum value for a short int
SHRT_MAX	32 767	Maximum value for a short int
USHRT_MAX	65 535	Maximum value for an unsigned short int
INT_MIN	–32 768	Minimum value for an int
INT_MAX	32 767	Maximum value for an int
UINT_MAX	65 535	Maximum value for an unsigned int
LONG_MIN	–2 147 483 648	Minimum value for a long int
LONG_MAX	2 147 483 647	Maximum value for a long int
ULONG_MAX	4 294 967 295	Maximum value for an unsigned long int
MB_LEN_MAX	1	Maximum number of bytes in multi-byte

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

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

Macro	Value	Description
FLT_RADIX	2	Base or radix of exponent representation
FLT_ROUNDS	1	Rounding mode for floating-point addition
FLT_DIG DBL_DIG LDBL_DIG	6 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 $1.0 + x \neq 1.0$
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. See the float.h header file supplied with the compiler for the full precision carried by the processor.

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

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

The math.h/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.

7.3.7 Nonlocal Jumps (setjmp.h/csetjmp)

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

- *jmp_buf*, an array type suitable for holding the information needed to restore a calling environment
- *setjmp*, a macro that saves its calling environment in its jmp_buf argument for later use by the longjmp function
- *longjmp*, a function that uses its jmp_buf argument to restore the program environment. The nonlocal jmp macro and function are listed in Table 7–3 (d) on page 7-27.

7.3.8 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 may vary each time a function is called.

The type, *va_list*, is a pointer type that can hold information for *va_start*, *va_end*, and *va_arg*.

A variable-argument function can use the macros declared by stdarg.h to step through its argument list at run time when the function 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. The variable argument functions are listed in Table 7–3 (e) page 7-27.

7.3.9 Standard Definitions (stddef.h/cstddef)

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

- The *ptrdiff_t* type, a signed integer type that is the data type resulting from the subtraction of two pointers
- The *size_t* type, an unsigned integer type that is the data type of the *sizeof* operator.

The macros include:

- The *NULL* macro, which expands to a null pointer constant (0)
- The *offsetof(type, identifier)* macro, which expands to an integer that has type *size_t*. The result is the value of an offset in bytes to a structure member (identifier) from the beginning of its structure (type).

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

7.3.10 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 include:

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

The macros include:

- The *NULL* macro, 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.
- The *BUFSIZ* macro, which expands to the size of the buffer that setbuf() uses.
- The *EOF* macro, which is the end-of-file marker.
- The *FOPEN_MAX* macro, which expands to the largest number of files that can be open at one time.

- The *FILENAME_MAX* macro, which expands to the length of the longest file name in characters.
- The *L_tmpnam* macro, which expands to the longest filename string that tmpnam() can generate.
- *SEEK_CUR*, *SEEK_SET*, and *SEEK_END*, macros that expand to indicate the position (current, start-of-file, or end-of-file, respectively) in a file.
- *TMP_MAX*, a macro that expands to the maximum number of unique filenames that tmpnam() can generate.
- *stderr*, *stdin*, *stdout*, which are pointers to the standard error, input, and output files, respectively.

The input/output functions are listed in Table 7–3 (f) on page 7-28.

7.3.11 General Utilities (stdlib.h/cstdlib)

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

- The *div_t* structure type that is the type of the value returned by the div function
- The *ldiv_t* 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 header also declares many of the common library functions:

- String conversion functions that convert strings to numeric representations
- Searching and sorting functions that allow you to search and sort arrays
- Sequence-generation functions that allow you to generate a pseudo-random sequence and allow you to choose a starting point for a sequence
- Program-exit functions that allow your program to terminate normally or abnormally
- Integer arithmetic that is not provided as a standard part of the C language

The general utility functions are listed in Table 7–3 (g) on page 7-30.

7.3.12 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, 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 string functions are listed in Table 7–3 (h) on page 7-32.

7.3.13 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.

The time functions and macros are listed in Table 7–3 (i) on page 7-33.

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 Central Standard Time, U.S.A.

The basis for all the functions in time.h 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. Since these functions and the CLOCKS_PER_SEC macro are system specific, only stubs are provided in the library. To use the other time functions, you must supply custom versions of these functions.

Note: Writing Your Own Clock Function

The clock function works with the stand-alone simulator. Used in this environment, clock() returns a cycle accurate count. The clock function returns –1 when used with the HLL debugger.

A host-specific clock function can be written. 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.

7.3.14 Exception Handling (exception and stdexcept)

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

7.3.15 Dynamic Memory Management (new)

The new header, which is 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.

7.3.16 Runtime Type Information (typeinfo)

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

7.4 Summary of Runtime-Support Functions and Macros

Table 7–3 summarizes the runtime-support header files (in alphabetical order) provided with the TMS320C55x ANSI C/C++ compiler. Most of the functions described are per the ANSI standard and behave exactly as described in the standard.

The functions and macros listed in Table 7–3 are described in detail in section 7.5, *Description of Runtime-Support Functions and Macros* on page 7-34. For a complete description of a function or macro, see the indicated page.

A superscripted number is used in the following descriptions to show exponents. For example, x^y is the equivalent of x to the power y.

Table 7–3. Summary of Runtime-Support Functions and Macros

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

Macro	Description	Page
void **assert**(int expr);	Inserts diagnostic messages into programs	7-38

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

Function	Description	Page
int **isalnum**(int c);	Tests c to see if it is an alphanumeric-ASCII character	7-58
int **isalpha**(int c);	Tests c to see if it is an alphabetic-ASCII character	7-58
int **isascii**(int c);	Tests c to see if it is an ASCII character	7-58
int **iscntrl**(int c);	Tests c to see if it is a control character	7-58
int **isdigit**(int c);	Tests c to see if it is a numeric character	7-58
int **isgraph**(int c);	Tests c to see if it is any printing character except a space	7-58
int **islower**(int c);	Tests c to see if it is a lowercase alphabetic ASCII charcter	7-58
int **isprint**(int c);	Tests c to see if it is a printable ASCII character (including a space)	7-58
int **ispunct**(int c);	Tests c to see if it is an ASCII punctuation character	7-58
int **isspace**(int c);	Tests c to see if it is an ASCII space bar, tab (horizontal or vertical), carriage return, form feed, or new line character	7-58
int **isupper**(int c);	Tests c to see if it is an uppercase ASCII alphabetic character	7-58
int **isxdigit**(int c);	Tests c to see if it is a hexadecimal digit	7-58
char **toascii**(int c);	Masks c into a legal ASCII value	7-93
char **tolower**(int char c);	Converts c to lowercase if it is uppercase	7-93
char **toupper**(int char c);	Converts c to uppercase if it is lowercase	7-93

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

Function	Description	Page
double **acos**(double x);	Returns the arc cosine of x	7-35
double **asin**(double x);	Returns the arc sine of x	7-38
double **atan**(double x);	Returns the arc tangent of x	7-39

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

Function	Description	Page
double **atan2**(double y, double x);	Returns the arc tangent of y/x	7-39
double **ceil**(double x);	Returns the smallest integer ≥ x; expands inline if –x is used	7-42
double **cos**(double x);	Returns the cosine of x	7-44
double **cosh**(double x);	Returns the hyperbolic cosine of x	7-44
double **exp**(double x);	Returns e^x	7-47
double **fabs**(double x);	Returns the absolute value of x	7-47
double **floor**(double x);	Returns the largest integer ≤ x; expands inline if –x is used	7-51
double **fmod**(double x, double y);	Returns the exact floating-point remainder of x/y	7-51
double **frexp**(double value, int *exp);	Returns f and exp such that .5 ≤ |f| < 1 and value is equal to $f \times 2^{exp}$	7-54
double **ldexp**(double x, int exp);	Returns $x \times 2^{exp}$	7-60
double **log**(double x);	Returns the natural logarithm of x	7-61
double **log10**(double x);	Returns the base-10 logarithm of x	7-61
double **modf**(double value, double *ip);	Breaks value into a signed integer and a signed fraction	7-67
double **pow**(double x, double y);	Returns x^y	7-68
double **sin**(double x);	Returns the sine of x	7-75
double **sinh**(double x);	Returns the hyperbolic sine of x	7-76
double **sqrt**(double x);	Returns the nonnegative square root of x	7-76
double **tan**(double x);	Returns the tangent of x	7-91
double **tanh**(double x);	Returns the hyperbolic tangent of x	7-91

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

Function or Macro	Description	Page
int **setjmp**(jmp_buf env);	Saves calling environment for use by longjmp; this is a macro	7-73
void **longjmp**(jmp_buf env, int _val);	Uses jmp_buf argument to restore a previously saved environment	7-73

(e) Variable argument macros (stdarg.h/cstdarg)

Macro	Description	Page
type **va_arg**(va_list, type);	Accesses the next argument of type type in a variable-argument list	7-94
void **va_end**(va_list);	Resets the calling mechanism after using va_arg	7-94
void **va_start**(va_list, parmN);	Initializes ap to point to the first operand in the variable-argument list	7-94

(f) 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 device table	7-35
void **clearerr**(FILE *_fp);	Clears the EOF and error indicators for the stream that _fp points to	7-43
int **fclose**(FILE *_fp);	Flushes the stream that _fp points to and closes the file associated with that stream	7-48
int **feof**(FILE *_fp);	Tests the EOF indicator for the stream that _fp points to	7-49
int **ferror**(FILE *_fp);	Tests the error indicator for the stream that _fp points to	7-49
int **fflush**(register FILE *_fp);	Flushes the I/O buffer for the stream that _fp points to	7-49
int **fgetc**(register FILE *_fp);	Reads the next character in the stream that _fp points to	7-50
int **fgetpos**(FILE *_fp, fpos_t *pos);	Stores the object that pos points to to the current value of the file position indicator for the stream that _fp points to	7-50
char ***fgets**(char *_ptr, register int _size, register FILE *_fp);	Reads the next _size minus 1 characters from the stream that _fp points to into array _ptr	7-50
FILE ***fopen**(const char *_fname, const char *_mode);	Opens the file that _fname points to; _mode points to a string describing how to open the file	7-52
int **fprintf**(FILE *_fp, const char *_format, ...);	Writes to the stream that _fp points to	7-52
int **fputc**(int _c, register FILE *_fp);	Writes a single character, _c, to the stream that _fp points to	7-52
int **fputs**(const char *_ptr, register FILE *_fp);	Writes the string pointed to by _ptr to the stream pointed to by _fp	7-53
size_t **fread**(void *_ptr, size_t _size, size_t _count, FILE *_fp);	Reads from the stream pointed to by _fp and stores the input to the array pointed to by _ptr	7-53
FILE ***freopen**(const char *_fname, const char *_mode, register FILE *_fp);	Opens the file that _fname points to using the stream that _fp points to; _mode points to a string describing how to open the file	7-54
int **fscanf**(FILE *_fp, const char *_fmt, ...);	Reads formatted input from the stream that _fp points to	7-55

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

Function	Description	Page
int **fseek**(register FILE *_fp, long _offset, int _ptrname);	Sets the file position indicator for the stream that _fp points to	7-55
int **fsetpos**(FILE *_fp, const fpos_t *_pos);	Sets the file position indicator for the stream that _fp points to to _pos. The pointer _pos must be a value from fgetpos() on the same stream.	7-55
long **ftell**(FILE *_fp);	Obtains the current value of the file position indicator for the stream that _fp points to	7-56
size_t **fwrite**(const void *_ptr, size_t _size, size_t _count, register FILE *_fp);	Writes a block of data from the memory pointed to by _ptr to the stream that _fp points to	7-56
int **getc**(FILE *_fp);	Reads the next character in the stream that _fp points to	7-56
int **getchar**(void);	A macro that calls fgetc() and supplies stdin as the argument	7-57
char ***gets**(char *_ptr);	Performs the same function as fgets() using stdin as the input stream	7-57
void **perror**(const char *_s);	Maps the error number in _s to a string and prints the error message	7-67
int **printf**(const char *_format, ...);	Performs the same function as fprintf but uses stdout as its output stream	7-68
int **putc**(int _x, FILE *_fp);	A macro that performs like fputc()	7-69
int **putchar**(int _x);	A macro that calls fputc() and uses stdout as the output stream	7-69
int **puts**(const char *_ptr);	Writes the string pointed to by _ptr to stdout	7-69
int **remove**(const char *_file);	Causes the file with the name pointed to by _file to be no longer available by that name	7-72
int **rename**(const char *_old_name, const char *_new_name);	Causes the file with the name pointed to by _old_name to be known by the name pointed to by _new_name	7-72
void **rewind**(register FILE *_fp);	Sets the file position indicator for the stream pointed to by _fp to the beginning of the file	7-72
int **scanf**(const char *_fmt, ...);	Performs the same function as fscanf() but reads input from stdin	7-73
void **setbuf**(register FILE *_fp, char *_buf);	Returns no value. setbuf() is a restricted version of setvbuf() and defines and associates a buffer with a stream	7-73
int **setvbuf**(register FILE *_fp, register char *_buf, register int _type, register size_t _size);	Defines and associates a buffer with a stream	7-75

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

Function	Description	Page
int **sprintf**(char *_string, const char *_format, ...);	Performs the same function as fprintf() but writes to the array that _string points to	7-76
int **sscanf** (const char *_str, const char *_fmt, ...);	Performs the same function as fscanf() but reads from the string that _str points to	7-77
FILE ***tmpfile**(void);	Creates a temporary file	7-92
char ***tmpnam**(char *_s);	Generates a string that is a valid filename (that is, the filename is not already being used)	7-92
int **ungetc**(int _c, register FILE *_fp);	Pushes the character specified by _c back into the input stream pointed to by _fp	7-94
int **vfprintf**(FILE *_fp, const char *_format, va_list _ap);	Performs the same function as fprintf() but replaces the argument list with _ap	7-95
int **vprintf**(const char *_format, va_list _ap);	Performs the same function as printf() but replaces the argument list with _ap	7-96
int **vsprintf**(char *_string, const char *_format, va_list _ap);	Performs the same function as sprintf() but replaces the argument list with _ap	7-96

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

Function	Description	Page
void **abort**(void);	Terminates a program abnormally	7-34
int **abs**(int i);	Returns the absolute value of val; expands inline unless –x0 is used	7-34
int **atexit**(void (*fun)(void));	Registers the function pointed to by fun, called without arguments at program termination	7-40
double **atof**(const char *st);	Converts a string to a floating-point value; expands inline if –x is used	7-40
int **atoi**(register const char *st);	Converts a string to an integer	7-40
long **atol**(register const char *st);	Converts a string to a long integer value; expands inline if –x is used	7-40
void ***bsearch**(register const void *key, register const void *base, size_t nmemb, size_t size, int (*compar)(const void *,const void *));	Searches through an array of nmemb objects for the object that key points to	7-41
void ***calloc**(size_t num, size_t size);	Allocates and clears memory for num objects, each of size bytes	7-42
div_t **div**(register int numer, register int denom);	Divides numer by denom producing a quotient and a remainder	7-46

(g) General functions (stdlib.h/cstdlib)(Continued)

Function	Description	Page
void **exit**(int status);	Terminates a program normally	7-47
void **free**(void *packet);	Deallocates memory space allocated by malloc, calloc, or realloc	7-53
char ***getenv**(const char *_string)	Returns the environment information for the variable associated with _string	7-57
long **labs**(long i);	Returns the absolute value of i; expands inline unless –x0 is used	7-34
ldiv_t **ldiv**(register long numer, register long denom);	Divides numer by denom	7-46
int **ltoa**(long val, char *buffer);	Converts val to the equivalent string	7-62
void ***malloc**(size_t size);	Allocates memory for an object of size bytes	7-62
void **minit**(void);	Resets all the memory previously allocated by malloc, calloc, or realloc	7-66
void **qsort**(void *base, size_t nmemb, size_t size, int (*compar) ());	Sorts an array of nmemb members; base points to the first member of the unsorted array, and size specifies the size of each member	7-70
int **rand**(void);	Returns a sequence of pseudorandom integers in the range 0 to RAND_MAX	7-70
void ***realloc**(void *packet, size_t size);	Changes the size of an allocated memory space	7-71
void **srand**(unsigned int seed);	Resets the random number generator	7-70
double **strtod**(const char *st, char **endptr);	Converts a string to a floating-point value	7-89
long **strtol**(const char *st, char **endptr, int base);	Converts a string to a long integer	7-89
unsigned long **strtoul**(const char *st, char **endptr**,** int base)**;**	Converts a string to an unsigned long integer	7-89

(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 cs; expands inline if –x is used	7-63
int **memcmp**(const void *cs, const void *ct, size_t n);	Compares the first n characters of cs to ct; expands inline if –x is used	7-63
void ***memcpy**(void *s1, const void *s2, register size_t n);	Copies n characters from s1 to s2	7-64
void ***memmove**(void *s1, const void *s2, size_t n);	Moves n characters from s1 to s2	7-64
void ***memset**(void *mem, register int ch, register size_t length);	Copies the value of ch into the first length characters of mem; expands inline if –x is used	7-64
char ***strcat**(char *string1, const char *string2);	Appends string2 to the end of string1	7-77
char ***strchr**(const char *string, int c);	Finds the first occurrence of character c in s; expands inline if –x is used	7-78
int **strcmp**(register const char *string1, register const char *s2);	Compares strings and returns one of the following values: <0 if string1 is less than string2; 0 if string1 is equal to string2; >0 if string1 is greater than string2. Expands inline if –x is used.	7-79
int **strcoll**(const char *string1, const char *string2);	Compares strings and returns one of the following values: <0 if string1 is less than string2; 0 if string1 is equal to string2; >0 if string1 is greater than string2.	7-79
char ***strcpy**(register char *dest, register const char *src);	Copies string src into dest; expands inline if –x is used	7-79
size_t **strcspn**(register const char *string, const char *chs);	Returns the length of the initial segment of string that is made up entirely of characters that are not in chs	7-80
char ***strerror**(int errno);	Maps the error number in errno to an error message string	7-81
size_t **strlen**(const char *string);	Returns the length of a string	7-83
char ***strncat**(char *dest, const char *src, register size_t n);	Appends up to n characters from src to dest	7-83
int **strncmp**(const char *string1, const char *string2, size_t n);	Compares up to n characters in two strings; expands inline if –x is used	7-85
char ***strncpy**(register char *dest, register const char *src, register size_t n);	Copies up to n characters from src to dest; expands inline if –x is used	7-86
char ***strpbrk**(const char *string, const char *chs);	Locates the first occurrence in string of *any* character from chs	7-87

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

Function	Description	Page
char ***strrchr**(const char *string, int c);	Finds the last occurrence of character c in string; expands inline if –x is used	7-87
size_t **strspn**(register const char *string, const char *chs);	Returns the length of the initial segment of string, which is entirely made up of characters from chs	7-88
char ***strstr**(register const char *string1, const char *string2);	Finds the first occurrence of string2 in string1	7-88
char ***strtok**(char *str1, const char *str2);	Breaks str1 into a series of tokens, each delimited by a character from str2	7-90
size_t **strxfrm**(register char *to, register const char *from, register size_t n);	Transforms n characters from from, to to	7-90

(i) Time-support functions (time.h/cstring)

Function	Description	Page
char ***asctime**(const struct tm *timeptr);	Converts a time to a string	7-37
clock_t **clock**(void);	Determines the processor time used	7-43
char ***ctime**(const time_t *timer);	Converts calendar time to local time	7-45
double **difftime**(time_t time1, time_t time0);	Returns the difference between two calendar times	7-45
struct tm ***gmtime**(const time_t *timer);	Converts local time to Greenwich Mean Time	7-58
struct tm ***localtime**(const time_t *timer);	Converts time_t value to broken down time	7-60
time_t **mktime**(register struct tm *tptr);	Converts broken down time to a time_t value	7-65
size_t **strftime**(char *out, size_t maxsize, const char *format, const struct tm *time);	Formats a time into a character string	7-81
time_t **time**(time_t *timer);	Returns the current calendar time	7-91

7.5 Description of Runtime-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 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 7-47.

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 i.

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

acos.c in rts.src

Description

The acos function returns the arc cosine of a floating-point argument x, which must be in the range [–1,1]. The return value is an angle in the range [0,π] radians.

Example

```
double realval, radians;

realval = 1.0;
radians = acos(realval);
return (radians);   /* acos return π/2 */
```

add_device — *Add Device to Device Table*

Syntax for C

```
#include <stdio.h>
int add_device(char *name,
               unsigned flags,
               int (*dopen)(),
               int (*dclose)(),
               int (*dread)(),
               int (*dwrite)(),
               fpos_t (*dlseek)(),
               int (*dunlink)(),
               int (*drename)());
```

Syntax for C++

```
#include <cstdio>
int std::add_device(char *name,
               unsigned flags,
               int (*dopen)(),
               int (*dclose)(),
               int (*dread)(),
               int (*dwrite)(),
               fpos_t (*dlseek)(),
               int (*dunlink)(),
               int (*drename)());
```

Defined in lowlev.c in rts.src

Description The add_device function adds a device record to the device table allowing that device to be used for input/output from C. The first entry in the device table is predefined to be the host device on which the debugger is running. The function add_device() finds the first empty position in the device table and initializes the fields of the structure that represent a device.

To open a stream on a newly-added device, use fopen() with a string of the format *devicename***:***filename* as the first argument.

- The *name* is a character string denoting the device name.
- The *flags* are device characteristics. The flags are as follows:

  _SSA Denotes that the device supports only one open stream at a time

  _MSA Denotes that the device supports multiple open streams

  More flags can be added by defining them in stdio.h.
- The dopen, dclose, dread, dwrite, dlseek, dunlink, drename specifiers are function pointers to the device drivers that are called by the low-level functions to perform I/O on the specified device. You must declare these functions with the interface specified in subsection 7.2.1, *Overview of Low-Level I/O Implementation*, on page 7-6. The device drivers for the host that the debugger is run on are included in the C I/O library.

Return Value The function returns one of the following values:

0 if successful

–1 if fails

Example This example does the following:

- Adds the device *mydevice* to the device table
- Opens a file named *test* on that device and associates it with the file **fid*
- Prints the string *Hello, world* into the file
- 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);
}
```

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 header declares and defines, see subsection 7.3.13, *Time Functions (time.h)*, on page 7-21.

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 π/2 */
```

assert — *Insert Diagnostic Information Macro*

Syntax for C

#include <assert.h>

void **assert**(int expr);

Syntax for C++

#include <cassert>

void **std::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;
```

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 */
```

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(rvalu, rvalv);    /* return value = 0 */
```

atexit — *Register Function Called by Exit ()*

Syntax for C

```
#include   <stdlib.h>

int atexit(void (*fun)(void));
```

Syntax for C++

```
#include <cstdlib>

int 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.

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

Description

Three functions convert strings to numeric representations:

- The atof function converts a string into a floating-point value. Argument st points to the string. The string must have the following format:

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

- The atoi function converts a string into an integer. Argument st points to the string; the string must have the following format:

  [*space*] [*sign*] *digits*

- The atol function converts a string into a long integer. Argument st points to the string. The string must have the following format:

  [*space*] [*sign*] *digits*

The *space* is indicated by a space (character), a horizontal or vertical tab, a carriage return, a form feed, or a new line. Following the *space* is an optional *sign*, and the *digits* that represent the integer portion of the number. The fractional part of the number follows, then the exponent, including an optional *sign*.

The first character that cannot be part of the number terminates the string.

The functions do not handle any overflow resulting from the conversion.

bsearch — *Array Search*

Syntax 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 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

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 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 2000 bytes. You can change this amount at link time by invoking the linker with the –heap option and specifying the desired size of the heap (in bytes) directly after the option. See subsection 6.1.5, *Dynamic Memory Allocation*, on page 6-6.

Example

This example uses the calloc routine to allocate and clear 20 bytes.

```
prt = calloc (10,2)  ;   /*Allocate and clear 20 bytes */
```

ceil — *Ceiling*

Syntax 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 */
```

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 in rts.src

Description

The clearerr function clears the EOF and error indicators for the stream that _fp points to.

clock — *Processor Time*

Syntax for C

#include <time.h>

clock_t **clock**(void);

Syntax for C++

#include <ctime>

void **std::clearerr**(FILE *_fp);

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.

cos — *Cosine*

Syntax for C

#include <math.h>

double **cos**(double x);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

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 */
```

cosh — *Hyperbolic Cosine*

Syntax for C

#include <math.h>

double **cosh**(double x);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in

cosh.c in rts.src

Description

The cosh function returns the hyperbolic cosine of a floating-point number x. A range error occurs if the magnitude of the argument is too large.

Example

```
double x, y;

x = 0.0;
y = cosh(x); /* return value = 1.0 */
```

ctime — *Calendar Time*

Syntax for C

#include <time.h>

char ***ctime**(const time_t *timer);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in

ctime.c in rts.src

Description

The ctime function converts a calendar time (pointed to by timer) to local time in the form of a string. This is equivalent to:

```
asctime(localtime(timer))
```

The function returns the pointer returned by the asctime function.

For more information about the functions and types that the time.h header declares and defines, see subsection 7.3.13, *Time Functions (time.h)*, on page 7-21.

difftime — *Time Difference*

Syntax for C

#include <time.h>

double **difftime**(time_t time1, time_t time0);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in

difftime.c in rts.src

Description

The difftime function calculates the difference between two calendar times, time1 minus time0. The return value expresses seconds.

For more information about the functions and types that the time.h header declares and defines, see subsection 7.3.13, *Time Functions (time.h)*, on page 7-21.

div/ldiv *Division*

Syntax for C

#include <stdlib.h>

div_t **div**(int numer, int denom);
ldiv_t **ldiv**(long numer, long denom);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in

div.c in rts.src

Description

Two functions support integer division by returning numer (numerator) divided by denom (denominator). You can use these functions to get both the quotient and the remainder in a single operation.

- The div function performs integer division. The input arguments are integers; the function returns the quotient and the remainder in a structure of type div_t. The structure is defined as follows:

```
typedef struct
{
   int  quot;         /*  quotient    */
   int  rem;          /* remainder    */
} div_t;
```

- The ldiv function performs long integer division. The input arguments are long integers; the function returns the quotient and the remainder in a structure of type ldiv_t. The structure is defined as follows:

```
typedef struct
{
   long int  quot;    /*  quotient    */
   long int  rem;     /* remainder    */
} ldiv_t;
```

The sign of the quotient is negative if either but not both of the operands is negative. The sign of the remainder is the same as the sign of the dividend.

exit — *Normal Termination*

Syntax for C

#include <stdlib.h>

void **exit**(int status);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in

exit.c in rts.src

Description

The exit function terminates a program normally. All functions registered by the atexit function are called in reverse order of their registration. The exit function can accept EXIT_FAILURE as a value. (See the abort function on page 7-34).

You can modify the exit function to perform application-specific shut-down tasks. The unmodified function simply runs in an infinite loop until the system is reset.

The exit function cannot return to its caller.

exp — *Exponential*

Syntax for C

#include <math.h>

double **exp**(double x);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in

exp.c in rts.src

Description

The exp function returns the exponential function of real number x. The return value is the number e raised to the power x. A range error occurs if the magnitude of x is too large.

Example

```
double x, y;

x = 2.0;
y = exp(x);        /* y = 7.38, which is e**2.0 */
```

fabs — *Absolute Value*

Syntax for C

#include <math.h>

double **fabs**(double x);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in

fabs.c in rts.src

Description

The fabs function returns the absolute value of a floating-point number, x.

Example

```
double x, y;

x = –57.5;
y = fabs(x);       /* return value = +57.5 */
```

fclose — *Close File*

Syntax for C

#include <stdio.h>

int **fclose**(FILE *_fp);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in

fclose.c in rts.src

Description

The fclose function flushes the stream that _fp points to and closes the file associated with that stream.

feof — *Test EOF Indicator*

Syntax for C

#include <stdio.h>

int **feof**(FILE *_fp);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in

feof.c in rts.src

Description

The feof function tests the EOF indicator for the stream pointed to by _fp.

ferror — *Test Error Indicator*

Syntax for C

#include <stdio.h>

int **ferror**(FILE *_fp);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in

ferror.c in rts.src

Description

The ferror function tests the error indicator for the stream pointed to by _fp.

fflush — *Flush I/O Buffer*

Syntax for C

#include <stdio.h>

int **fflush**(register FILE *_fp);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in

fflush.c in rts.src

Description

The fflush function flushes the I/O buffer for the stream pointed to by _fp.

fgetc — *Read Next Character*

Syntax for C

#include <stdio.h>

int **fgetc**(register FILE *_fp);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in fgetc.c in rts.src

Description The fgetc function reads the next character in the stream pointed to by _fp.

fgetpos — *Store Object*

Syntax for C

#include <stdio.h>

int **fgetpos**(FILE *_fp, fpos_t *pos);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in fgetpos.c in rts.src

Description The fgetpos function stores the object pointed to by pos to the current value of the file position indicator for the stream pointed to by _fp.

fgets — *Read Next Characters*

Syntax for C

#include <stdio.h>

char ***fgets**(char *_ptr, register int _size, register FILE *_fp);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in fgets.c in rts.src

Description The fgets function reads the specified number of characters from the stream pointed to by _fp. The characters are placed in the array named by _ptr. The number of characters read is _size –1.

floor — *Floor*

Syntax for C

#include <math.h>

double **floor**(double x);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in

floor.c in rts.src

Description

The floor function returns a floating-point number that represents the largest integer less than or equal to x.

Example

```
double answer;

answer = floor(3.1415);      /* answer =  3.0 */
answer = floor(–3.5);        /* answer = –4.0 */
```

fmod — *Floating-Point Remainder*

Syntax for C

#include <math.h>

double **fmod**(double x, double y);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in

fmod.c in rts.src

Description

The fmod function returns the floating-point remainder of x divided by y. If y == 0, the function returns 0.

Example

```
double x, y, r;

x = 11.0;
y = 5.0;
r = fmod(x, y);       /* fmod returns 1.0 */
```

fopen — *Open File*

Syntax for C

#include <stdio.h>

FILE ***fopen**(const char *_fname, const char *_mode);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in

fopen.c in rts.src

Description

The fopen function opens the file that _fname points to. The string pointed to by _mode describes how to open the file.

fprintf — *Write Stream*

Syntax for C

#include <stdio.h>

int fprintf(FILE *_fp, const char *_format, ...**);**

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

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.

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.

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.

fread — *Read Stream*

Syntax for C

#include <stdio.h>

size_t **fread**(void *_ptr, size_t size, size_t count, FILE *_fp);

Syntax for C++

#include <cstdio>

size_t **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.

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 function takes no action and returns. For more information, see subsection 6.1.5, *Dynamic Memory Allocation*, on page 6-6.

Example

This example allocates ten bytes and then frees them.

```
char *x;
x = malloc(10);       /*  allocate 10 bytes   */
free(x);              /*  free 10 bytes       */
```

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.

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 in rts.src

Description

The frexp function breaks a floating-point number into a normalized fraction and the integer power of 2. The function returns a value with a magnitude in the range [1/2, 1] or 0, so that value = = $x \times 2^{exp}$. The frexp function stores the power in the int pointed to by exp. If value is 0, both parts of the result are 0.

Example

```
double fraction;
int exp;

fraction = frexp(3.0, &exp);
/* after execution, fraction is .75 and exp is 2 */
```

fscanf — *Read Stream*

Syntax 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.

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.

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.

ftell — *Get Current File Position Indicator*

Syntax for C

#include <stdio.h>

long **ftell**(FILE *_fp);

Syntax for C++

#include <cstdio>

long **ftell**(FILE *_fp);

Defined in

ftell.c in rts.src

Description

The ftell function gets the current value of the file position indicator for the stream pointed to by _fp.

fwrite — *Write Block of Data*

Syntax 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.

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.

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.

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 trgdrv.c in rts.src

Description The getenv function returns the environment information for the variable associated with _string.

gets — *Read Next From Standard Input*

Syntax 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.

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 header declares and defines, see subsection 7.3.13, *Time Functions (time.h)*, on page 7-21.

isxxx — *Character Typing*

Syntax for C

#include <ctype.h>

int **isalnum**(int c);
int **isalpha**(int c);
int **isascii**(int c);
int **iscntrl**(int c);
int **isdigit**(int c);
int **isgraph**(int c);
int **islower**(int c);
int **isprint**(int c);
int **ispunct**(int c);
int **isspace**(int c);
int **isupper**(int c);
int **isxdigit**(int c);

Syntax for C++

#include <cctype>

int **std::isalnum**(int c);
int **std::isalpha**(int c);
int **std::isascii**(int c);
int **std::iscntrl**(int c);
int **std::isdigit**(int c);
int **std::isgraph**(int c);
int **std::islower**(int c);
int **std::isprint**(int c);
int **std::ispunct**(int c);
int **std::isspace**(int c);
int **std::isupper**(int c);
int **std::isxdigit**(int c);

Defined in

isxxx.c and ctype.c in rts.src
Also defined in ctype.h/cctype as macros

Description

These functions test a single argument c to see if it is a particular type of character — alphabetic, alphanumeric, numeric, ASCII, etc. If the test is true, the function returns a nonzero value; if the test is false, the function returns 0. The character typing functions include:

isalnum	Identifies alphanumeric ASCII characters (tests for any character for which isalpha or isdigit is true)
isalpha	Identifies alphabetic ASCII characters (tests for any character for which islower or isupper is true)
isascii	Identifies ASCII characters (any character from 0–127)
iscntrl	Identifies control characters (ASCII characters 0–31 and 127)
isdigit	Identifies numeric characters between 0 and 9 (inclusive)
isgraph	Identifies any nonspace character
islower	Identifies lowercase alphabetic ASCII characters
isprint	Identifies printable ASCII characters, including spaces (ASCII characters 32–126)
ispunct	Identifies ASCII punctuation characters
isspace	Identifies ASCII tab (horizontal or vertical), space bar, carriage return, form feed, and new line characters
isupper	Identifies uppercase ASCII alphabetic characters
isxdigit	Identifies hexadecimal digits (0–9, a–f, A–F)

The C compiler also supports a set of macros that perform these same functions. The macros have the same names as the functions but are prefixed with an underscore; for example, _isascii is the macro equivalent of the isascii function. In general, the macros execute more efficiently than the functions.

labs

See abs/labs on page 7-34.

ldexp — *Multiply by a Power of Two*

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 */
```

ldiv

See div/ldiv on page 7-46.

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 header declares and defines, see subsection 7.3.13, *Time Functions (Time.h)* on page 7-21.

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 */
```

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 */
```

longjmp — *See setjmp/longjmp on page 7-73.*

ltoa — *Long Integer to ASCII*

Syntax for C

no prototype provided

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

Syntax for C++

no prototype provided

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

Defined in

ltoa.c in rts.src

Description

The ltoa function is a nonstandard (non-ANSI) function and is provided for compatibility. 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.

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 2000 bytes. You can change this amount at link time by invoking the linker with the –heap option and specifying the desired size of the heap (in bytes) directly after the option. For more information, see subsection 6.1.5, *Dynamic Memory Allocation*, on page 6-6.

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 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.

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 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.

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 ***memcpy**(void *s1, const void *s2, size_t n);

Defined in

memcpy.c in rts.src

Description

The memcpy function copies n characters from the object that s2 points to into the object that s1 points to. If you attempt to copy characters of overlapping objects, the function's behavior is undefined. The function returns the value of s1.

The memcpy function is similar to strncpy, except that the objects that memcpy copies can contain values of 0.

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 in rts.src

Description

The memmove function moves n characters from the object that s2 points to into the object that s1 points to; the function returns the value of s1. The memmove function correctly copies characters between overlapping objects.

memset — *Duplicate Value in Memory*

Syntax 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 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.

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 header declares and defines, see subsection 7.3.13, *Time Functions (time.h)*, on page 7-21.

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 */
```

minit — *Reset Dynamic Memory Pool*

Syntax for C

no prototype provided

void **minit**(void);

Syntax for C++

no prototype provided

void **std::minit**(void);

Defined in memory.c in rts.src

Description The minit function resets all the space that was previously allocated by calls to the malloc, calloc, or realloc functions.

The memory that minit uses is in a special memory pool or heap. The constant _ _SYSMEM_SIZE defines the size of the heap as 2000 bytes. You can change this amount at link time by invoking the linker with the –heap option and specifying the desired size of the heap (in bytes) directly after the option. For more information, refer to subsection 6.1.5, *Dynamic Memory Allocation*, on page 6-6.

Note: No Previously Allocated Objects are Available After minit

Calling the minit function makes *all* the memory space in the heap available again. Any objects that you allocated previously will be lost; do not try to access them.

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 in rts.src

Description

The modf function breaks a value into a signed integer and a signed fraction. Each of the two parts has the same sign as the input argument. The function returns the fractional part of 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.           */
```

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.

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 ≤ 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 */
```

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.

putc — *Write Character*

Syntax for C

#include <stdio.h>

int **putc**(int _x, FILE *_fp);

Syntax for C++

#include <cstdio>

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.

putchar — *Write Character to Standard Output*

Syntax for C

#include <stdio.h>

int **putchar**(int _x);

Syntax for C++

#include <cstdio>

int **std::putchar**(int _x);

Defined in

putchar.c in rts.src

Description

The putchar function writes a character to the standard output device.

puts — *Write to Standard Output*

Syntax for C

#include <stdio.h>

int **puts**(const char *_ptr);

Syntax for C++

#include <cstdio>

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.

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 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

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.

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 calloc uses is in a special memory pool or heap. The constant _ _SYSMEM_SIZE defines the size of the heap as 2000 bytes. You can change this amount at link time by invoking the linker with the –heap option and specifying the desired size of the heap (in bytes) directly after the option. For more information, see subsection 6.1.5, *Dynamic Memory Allocation*, on page 6-6.

remove — *Remove File*

Syntax for C

#include <stdio.h>

int **remove**(const char *_file);

Syntax for C++

#include <cstdio>

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.

rename — *Rename File*

Syntax for C

#include <stdio.h>

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

Syntax for C++

#include <cstdio>

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.

rewind — *Position File Position Indicator to Beginning of File*

Syntax for C

#include <stdio.h>

void **rewind**(register FILE *_fp);

Syntax for C++

#include <cstdio>

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.

scanf — *Read Stream From Standard Input*

Syntax for C

```
#include   <stdio.h>

int scanf(const char *_fmt, ...);
```

Syntax for C++

```
#include <cstdio>

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.

setbuf — *Specify Buffer for Stream*

Syntax for C

```
#include   <stdio.h>

void setbuf(register FILE *_fp, char *_buf);
```

Syntax for C++

```
#include <cstdio>

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.

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 setjmp.asm in rts.src

Description The setjmp.h header defines one type, one macro, and one function for bypassing the normal function call and return discipline:

- The **jmp_buf** type is an array type suitable for holding the information needed to restore a calling environment.
- The **setjmp** macro saves its calling environment in the jmp_buf argument for later use by the longjmp function.

  If the return is from a direct invocation, the setjmp macro returns the value 0. If the return is from a call to the longjmp function, the setjmp macro returns a nonzero value.
- The **longjmp** function restores the environment that was saved in the jmp_buf argument by the most recent invocation of the setjmp macro. If the setjmp macro was not invoked, or if it terminated execution irregularly, the behavior of longjmp is undefined.

  After longjmp is completed, the program execution continues as if the corresponding invocation of setjmp had just returned 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.

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);
   . . .
}
```

setvbuf — *Define and Associate Buffer With Stream*

Syntax for C

#include <stdio.h>

int **setvbuf**(register FILE *_fp, register char *_buf, register int _type,
register size_t _size);

Syntax for C++

#include <cstdio>

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

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  */
```

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 */
```

sprintf — *Write Stream*

Syntax for C

#include <stdio.h>

int **sprintf**(char _string, const char *_format, ...);

Syntax for C++

#include <cstdio>

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.

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 */
```

srand

See rand/srand on page 7-70.

sscanf

Read Stream

Syntax for C

#include <stdio.h>

int **sscanf**(const char *str, const char *format, ...);

Syntax for C++

#include <cstdio>

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.

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 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"                                */
```

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 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.

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 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.

The functions return one of the following values:

< 0	if *string1 is less than *string2
0	if *string1 is equal to *string2
> 0	if *string1 is greater than *string2

Example

```
char *stra = ”why ask why”;
char *strb = ”just do it”;
char *strc = ”why ask why”;

if (strcmp(stra, strb) > 0)
   {
      /*    statements here will be executed    */
   }
if (strcoll(stra, strc) == 0)
   {
      /* statements here will be executed also */
   }
```

strcpy — *String Copy*

Syntax for C

#include <string.h>

char ***strcpy**(char *dest, const char *src);

Syntax for C++

#include <cstring>

char ***std::strcpy**(char *dest, const char *src);

Defined in

strcpy.c in rts.src

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

Example In the following example, the strings pointed to by *a and *b are two separate and distinct memory locations. In the comments, the notation \0 represents the null character:

```
char *a = "The quick black fox";
char *b = " jumps over ";

/* a --> "The quick black fox\0"   */
/* b --> " jumps over \0"          */

strcpy(a,b);

/* a --> " jumps over \0"          */
/* b --> " jumps over \0"          */
```

strcspn — *Find Number of Unmatching Characters*

Syntax 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 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 */
```

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 in rts.src

Description The strerror function returns the string “string error”. This function is supplied to provide ANSI compatibility.

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 header declares and defines, see subsection 7.3.13, *Time Functions (time.h)*, on page 7-21.

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 in rts.src

Description

The strlen function returns the length of string. In C, a character string is terminated by the first byte with a value of 0 (a null character). The returned result does not include the terminating null character.

Example

```
char *stra = ”who is there?”;
char *strb = ”abcdefghijklmnopqrstuvwxyz”;
char *strc = ”abcdefg”;
size_t length;

length = strlen(stra);      /* length = 13 */
length = strlen(strb);      /* length = 26 */
length = strlen(strc);      /* length = 7  */
```

strncat — *Concatenate Strings*

Syntax 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 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"              */;
```

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 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 (strcmp(stra, strb, size) > 0)
   {
      /* statements here will get executed */
   }
if (strcomp(stra, strc, size) == 0)
   {
      /* statements here will get executed also */
   }
```

strncpy — *String Copy*

Syntax for C

#include <string.h>

char ***strncpy**(char *dest, const char *src, size_t n);

Syntax for C++

#include <cstring>

char ***std::strncpy**(char *dest, const char *src, size_t n);

Defined in

strncpy.c in rts.src

Description

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

Example

Note that strb contains a leading space to make it five characters long. Also note that the first five characters of strc are an I, a space, the word am, and another space, so that after the second execution of strncpy, stra begins with the phrase I am followed by two spaces. In the comments, the notation \0 represents the null character.

```
char *stra = ”she’s the one mother warned you of”;
char *strb = ” he’s”;
char *strc = ”I am the one father warned you of”;
char *strd = ”oops”;
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”                               */;
```

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 in rts.src

Description

The strpbrk function locates the first occurrence in string of *any* character in chs. If strpbrk finds a matching character, it returns a pointer to that character; otherwise, it returns a null pointer (0).

Example

```
char *stra = ”it wasn’t me”;
char *strb = ”wave”;
char *a;

a = strpbrk (stra,strb);
```

After this example, *a points to the w in wasn’t.

strrchr — *Find Last Occurrence of a Character*

Syntax 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 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.

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 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 = strspn(stra,strb);     /* length = 3 */
length = strspn(stra,strc);     /* length = 0 */
```

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 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 doesn't find the string, it returns a null pointer. If string2 points to a string with length 0, strstr returns string1.

Example

```
char *stra = "so what do you want for nothing?";
char *strb = "what";
char *ptr;

ptr = strstr(stra,strb);
```

The pointer *ptr now points to the w in what in the first string.

strtod/strtol/ strtoul

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

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 what base to interpret the string in.

- The strtod function converts a string to a floating-point value. The string must have the following format:

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

  The function returns the converted string; if the original string is empty or does not have the correct format, the function returns a 0. If the converted string would cause an overflow, the function returns ±HUGE_VAL; if the converted string would cause an underflow, the function returns 0. If the converted string overflows or an 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.

strtok — *Break String into Token*

Syntax for C

#include <string.h>

char ***strtok**(char *str1, const char *str2);

Syntax for C++

#include <cstdio>

char ***std::strtok**(char *str1, const char *str2);

Defined in

strtok.c in rts.src

Description

Successive calls to the strtok function break str1 into a series of tokens, each delimited by a character from str2. Each call returns a pointer to the next token.

Example

After the first invocation of strtok in the example below, the pointer stra points to the string excuse\0 because strtok has inserted a null character where the first space used to be. In the comments, the notation \0 represents the null character.

```
char *stra = ”excuse me while I kiss the sky”;
char *ptr;

ptr = strtok (stra,” ”); /* ptr –> ”excuse\0” */
ptr = strtok (0,” ”);    /* ptr –> ”me\0”     */
ptr = strtok (0,” ”);    /* ptr –> ”while\0”  */
```

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);

Description

The strxfrm function converts n characters pointed to by from into the n characters pointed to by to.

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 */
```

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 */
```

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 header declares and defines, see subsection 7.3.13, *Time Functions (time.h)* page 7-21.

> **Note: The time Function Is Target-System Specific**
>
> The time function is target-system specific, so you must write your own time function.

tmpfile — *Create Temporary File*

Syntax for C

```
#include   <stdio.h>

FILE *tmpfile(void);
```

Syntax for C++

```
#include <cstdio>

FILE *std::tmpfile(void);
```

Defined in tmpfile.c in rts.src

Description The tmpfile function creates a temporary file.

tmpnam — *Generate Valid Filename*

Syntax for C

```
#include   <stdio.h>

char *tmpnam(char *_s);
```

Syntax for C++

```
#include <cstdio>

char *std::tmpnam(char *_s);
```

Defined in tmpnam.c in rts.src

Description The tmpnam function generates a string that is a valid filename.

toascii — *Convert to ASCII*

Syntax for C

```
#include    <ctype.h>

int toascii(int c);
```

Syntax for C++

```
#include <cctype>

int toascii(int c);
```

Defined in

toascii.c in rts.src

Description

The toascii function ensures that c is a valid ASCII character by masking the lower seven bits. There is also an equivalent macro call _toascii.

tolower/toupper — *Convert Case*

Syntax 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 upper case or lower case:

- 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.

ungetc — *Write Character to Stream*

Syntax for C

```
#include   <stdio.h>

int ungetc(int c, FILE *_fp);
```

Syntax for C++

```
#include <cstdio>

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.

va_arg/va_end/va_start — *Variable-Argument Macros*

Syntax for C

```
#include   <stdarg.h>

typedef    char *va_list;
           type va_arg(va_list, _type);
           void va_end(va_list);
           void va_start(va_list, parmN);
```

Syntax for C++

```
#include <cstdarg>

typedef    char *std::va_list;
           type std::va_arg(va_list, _type);
           void std::va_end(va_list);
           void std::va_start(va_list, parmN);
```

Defined in

stdarg.h/cstdarg

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 right-most parameter in the fixed, declared list.
- The va_arg macro returns the value of the next argument in a call to a variable-argument function. Each time you call va_arg, it modifies _ap so that successive arguments for the variable-argument function can be returned by successive calls to va_arg (va_arg modifies _ap to point to the next argument in the list). The type parameter is a type name; it is the type of the current argument in the list.
- The va_end macro resets the stack environment after va_start and va_arg are used.

Note that 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                    */
}
```

vfprintf — *Write to Stream*

Syntax for C

#include <stdio.h>

int **vfprintf**(FILE *_fp, const char *_format, char *_ap);

Syntax for C++

#include <cstdio>

int **std::vfprintf**(FILE *_fp, const char *_format, char *_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.

vprintf — *Write to Standard Output*

Syntax for C

#include <stdio.h>

int **vprintf**(const char *_format, char *_ap);

Syntax for C++

#include <cstdio>

int **std::vprintf**(const char *_format, char *_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.

vsprintf — *Write Stream*

Syntax for C

#include <stdio.h>

int **vsprintf**(char *string, const char *_format, char *_ap);

Syntax for C++

#include <cstdio>

int **std::vsprintf**(char *string, const char *_format, char *_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.

Chapter 8

Library-Build Utility

When using the TMS320C55x™ 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 runtime-support libraries, this package includes the source archive, rts.src, which contains all runtime-support functions.

You can build your own runtime-support libraries by using the mk55 utility described in this chapter and the archiver described in the *TMS320C55x Assembly Language Tools User's Guide*.

Topic **Page**

8.1 Invoking the Library-Build Utility

The syntax for invoking the library-build utility is:

mk55 [*options*] *src_arch1* [**–l***obj.lib1*] [*src_arch2* [**–l***obj.lib2*]] ...

mk55 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 Sections 8.2 and 8.3.)

src_arch The name of a source archive file. For each source archive named, mk55 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, mk55 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 mk55 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 C55X_C_OPTION, C_OPTION, C55X_C_DIR, and C_DIR.

8.2 Library-Build Utility Options

Most of the options that are included on the command line correspond directly to options of the same name used with the compiler, assembler, linker, and shell. The following options apply only to the library-build utility.

––c Extracts C source files contained in the source archive from the library and leaves them in the current directory after the utility 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 runtime-support header files from the rts.src archive that is shipped with the tools.

––k Overwrite files. By default, the utility aborts any time it attempts to create an object file when an object file of the same name already exists in the current directory, regardless of whether you specified the name or the utility derived it.

––q Suppress header information (quiet).

––u Does not use the header files contained in the source archive when building the object library. If the desired headers are already in the current directory, there is no reason to reinstall them. This option gives you flexibility in modifying runtime-support functions to suit your application.

––v Prints progress information to the screen during execution of the utility. Normally, the utility operates silently (no screen messages).

8.3 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 8–1 lists these options. These options are described in detail on the indicated page below.

Table 8–1. Summary of Options and Their Effects

(a) Options that control the compiler/shell

Option	Effect	Page
–g	Enables symbolic debugging	2-14

(b) Options that control the parser

Option	Effect	Page
–pi	Disables definition-controlled inlining (but –o3 optimizations still perform automatic inlining)	2-38
–pk	Makes code K&R compatible	5-31
–pr	Enables relaxed mode; ignores strict ANSI violations	5-31
–ps	Enables strict ANSI mode (for C, not K&R C)	5-31

(c) Options that control diagnostics

Option	Effect	Page
–pdr	Issues remarks (nonserious warnings)	2-31
–pdv	Provides verbose diagnostics that display the original source with line wrap	2-32
–pdw	Suppresses warning diagnostics (errors are still issued)	2-32

(d) Options that control the optimization level

Option	Effect	Page
–o0	Compiles with register optimization	3-2
–o1	Compiles with –o0 optimization + local optimization	3-2
–o2 (or –o)	Compiles with –o1 optimization + global optimization	3-3
–o3	Compiles with –o2 optimization + file optimization. Note that mk55 automatically sets –oI0 and –op0.	3-3

Table 8–1. Summary of Options and Their Effects (Continued)

(e) Options that are target-specific

Option	Effect	Page
–ma	Assumes variables are aliased	3-11
–mg	Assumes algebraic assembly files	2-15
–mn	Enables optimizer options disabled by –g	3-15

(f) Option that controls the assembler

Option	Effect	Page
–as	Keeps labels as symbols	2-21

(g) Options that change the default file extensions

Option	Effect	Page
–ea[.]*extension*	Sets default extension for assembly files	2-19
–eo[.]*extension*	Sets default extension for object files	2-19

Chapter 9

C++ Name Demangler

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.

These topics 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**

9.1 Invoking the C++ Name Demangler

The syntax for invoking the C++ name demangler is:

dem55 [*options*][*filenames*]

dem55	Command that invokes the C++ name demangler.
options	Options affect how the name demangler behaves. Options can appear anywhere on the command line or in a linker command file. (Options are discussed in section 9.2.)
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, dem55 uses standard in.

By default, the C++ name demangler sends output to standard out. You can use the –o file option if you want to send output to a file.

9.2 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*	Sends output to the given file rather than to standard out
–u	Specifies that external names do not have a C++ prefix
–v	Enables verbose mode (outputs a banner)

9.3 Sample Usage of the C++ Name Demangler

Example 9–1 shows a sample C++ program and the resulting assembly that is output by the TMS320C55x™ compiler. In Example 9–1(b), the linknames of foo( ) and compute( ) are mangled; that is, their signature information is encoded into their names.

Example 9–1. 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((int)val, &last_err);

   *err = last_err;
   return result;
}
```

(b) Partial resulting assembly for foo()

```
;*************************************************************
;* FUNCTION NAME: _foo(short, int *)                         *
;*************************************************************
___foo__FsPi:
            .bss  _last_err$1,1,0,0
            AADD #-3, SP
            MOV *abs16(#_last_err$1), AR1
            MOV AR0, *SP(#1)
            MOV T0, *SP(#0)
            MOV #0, *SP(#2)
            BCC L1, AR1 != #0
                                    ; branch occurs
            MOV #_last_err$1, AR0
            call #___compute__FsPi   ; call occurs
            MOV T0, *SP(#2)
L1:
            MOV *SP(#1), AR3
            MOV *abs16(#_last_err$1), AR1
            MOV AR1, *AR3
            MOV *SP(#2), T0
            AADD #3, SP
            return
```

Executing the C++ name demangler utility demangles all names that it believes to be mangled. If you enter:

```
% dem55 foo.asm
```

the result is shown in Example 9–2. Notice that the linknames of foo( ) and compute( ) are demangled.

Example 9–2. Result After Running the C++ Name Demangler Utility

```
;*********************************************************
;* FUNCTION NAME: foo(short, int *)                      *
;*********************************************************
foo(short, int *):
            .bss  _last_err$1,1,0,0
            AADD #-3, SP
            MOV *abs16(#_last_err$1), AR1
            MOV AR0, *SP(#1)
            MOV T0, *SP(#0)
            MOV #0, *SP(#2)
            BCC L1, AR1 != #0
                                    ; branch occurs
            MOV #_last_err$1, AR0
            call #compute(short, int *)   ; call occurs
            MOV T0, *SP(#2)
L1:
            MOV *SP(#1), AR3
            MOV *abs16(#_last_err$1), AR1
            MOV AR1, *AR3
            MOV *SP(#2), T0
            AADD #3, SP
            return
```

Appendix A

Glossary

A

ANSI: American National Standards Institute. An organization that establishes standards voluntarily followed by industries.

alias disambiguation: A technique that determines when two pointer expressions cannot point to the same location, allowing the compiler to freely optimize such expressions.

aliasing: Aliasing occurs when a single object can 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 refer to any other object.

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

archive library: A collection of individual files grouped into a single file by the archiver.

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

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

assignment statement: A statement that initializes a variable with a value.

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

autoinitialization at load time: An autoinitialization method used by the linker when linking C code. The linker uses this method when you invoke the linker with the –cr option. This method initializes variables at load time instead of runtime.

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

B

big-endian: An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *little endian*

block: A set of statements that are grouped together with braces and treated as an entity.

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

:byte: The smallest addressable unit of storage that can contain a character.

C

C compiler: A software program that translates C source statements into assembly language source statements.

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

command file: A file that contains linker or hex conversion utility options and names input files for the linker or hex conversion utility.

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

common object file format (COFF): A binary object file format that promotes modular programming by supporting the concept of *sections*. All COFF sections are independently relocatable in memory space; you can place any section into any allocated block of target memory.

constant: A type whose value cannot change.

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

D

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

direct call: A function call where one function calls another using the function's name.

directives: Special-purpose commands that control the actions and functions of a software tool.

disambiguation: See *alias disambiguation*

dynamic memory allocation: A technique used by several functions (such as malloc, calloc, and realloc) to dynamically allocate memory for variables at runtime. This is accomplished by defining a large memory pool (heap) and using the functions to allocate memory from the heap.

E

emulator: A development system used to test software directly on TMS320C55x™ hardware.

entry point: A point in target memory where execution starts.

environment variable: System symbol that you define and assign to a string. They are often included in batch files, for example, .cshrc.

epilog: The portion of code in a function that restores the stack and returns.

executable module: A linked object file that can be executed in a target system.

expression: A constant, a symbol, or a series of constants and symbols separated by arithmetic operators.

external symbol: A symbol that is used in the current program module but defined or declared in a different program module.

F

file-level optimization: A level of optimization where the compiler uses the information that it has about the entire file to optimize your code (as opposed to program-level optimization, where the compiler uses information that it has about the entire program to optimize your code).

function inlining: The process of inserting code for a function at the point of call. This saves the overhead of a function call, and allows the optimizer to optimize the function in the context of the surrounding code.

G

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

I

indirect call: A function call where one function calls another function by giving the address of the called function.

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

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

interlist utility: A utility that inserts as comments your original C source statements into the assembly language output from the assembler. The C statements are inserted next to the equivalent assembly instructions.

K

kernel: The body of a software-pipelined loop between the pipelined-loop prolog and the pipelined-loop epilog.

K&R C: Kernighan and Ritchie C, the de facto standard as defined in the first edition of *The C Programming Language* (K&R). Most K&R C programs written for earlier, non-ANSI C compilers 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. A label is the only assembler statement that can begin in column 1.

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

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

little endian: An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *big endian*

loader: A device that loads an executable module into system memory.

loop unrolling: An optimization that expands small loops so that each iteration of the loop appears in your code. Although loop unrolling increases code size, it can improve the efficiency of your code.

M

macro: A user-defined routine that can be used as an instruction.

macro call: The process of invoking a macro.

macro definition: A block of source statements that define the name and the code that make up a macro.

macro expansion: The process of inserting source statements into your code in place of a macro call.

map file: An output file, created by the linker, that shows the memory configuration, section composition, section allocation, symbol definitions, and the addresses at which the symbols were defined for your program.

memory map: A map of target system memory space that is partitioned into functional blocks.

O

object file: An assembled or linked file that contains machine-language object code.

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

operand: An argument of an assembly language instruction, assembler directive, or macro directive that supplies information to the operation performed by the instruction or directive.

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

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

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

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

P

parser: A software tool that reads the source file, performs preprocessing functions, checks the syntax, and produces an intermediate file that can be used as input for the optimizer or code generator.

pragma: Preprocessor directive that provides directions to the compiler about how to treat a particular statement.

preprocessor: A software tool that interprets macro definitions, expands macros, interprets header files, interprets conditional compilation, and acts upon preprocessor directives.

program-level optimization: An aggressive level of optimization where all of the source files are compiled into one intermediate file. Because the compiler can see the entire program, several optimizations are performed with program-level optimization that are rarely applied during file-level optimization.

R

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

runtime environment: The runtime parameters in which your program must function. These parameters are defined by the memory and register conventions, stack organization, function call conventions, and system initialization.

runtime-support functions: Standard ANSI functions that perform tasks that are not part of the C language (such as memory allocation, string conversion, and string searches).

runtime-support library: A library file, rts.src, that contains the source for the runtime-support functions.

S

section: A relocatable block of code or data that ultimately occupies 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.

shell program: A utility that lets you compile, assemble, and optionally link in one step. The shell runs one or more source modules through the compiler (including the parser, optimizer, and code generator), the assembler, and the linker.

:simulator: A development system used to test software on a workstation without C55x hardware.

source file: A file that contains C code or assembly language code that is compiled or assembled to form an object file.

standalone preprocessor: A software tool that expands macros, #include files, and conditional compilation as an independent program. It also performs integrated preprocessing, which includes parsing of instructions.

static variable: A 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: Any entry in the symbol table that indicates how to access a symbol.

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

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

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

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

T

target system: The system on which the object code you have developed is executed.

.text section: One of the default COFF sections. The .text section is initialized and contains executable code. You can use the .text directive to assemble code into the .text section.

trigraph sequence: A three character sequence that has a meaning (as defined by the ISO 646-1983 Invariant Code Set). These characters cannot be represented in the C character set and are expanded to one character. For example, the trigraph ??' is expanded to ^.

U

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

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

V

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

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z