MSP430
Optimizing C/C++ Compiler
v 2.0.1
User's Guide

Literature Number: SLAU132A
June 2006

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

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. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

Preface

Read This First

About This Manual

The *MSP430 Optimizing C/C++ Compiler User's Guide* explains how to use these compiler tools:

- Compiler
- Interlist utility
- Library-build process
- C++ name demangling utility

The MSP430 C/C++ compiler accepts C and C++ code conforming to the American National Standards Institute (ANSI) and International Organization for Standardization (ISO) standards for these languages, and produces assembly language source code for the MSP430 device. The compiler supports the 1989 version of the C language.

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

Notational Conventions

This document uses the following conventions:

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

  Here is a sample of C code where the #define directive is emphasized:

```
#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:

  cl430 [*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 do not enter the brackets themselves. This is an example of a command that has optional parameters:

  cl430 [*options*] [*filenames*] [−z [*link_options*] [*object files*]]

  The cl430 command has several optional parameters.

- Braces ({ and }) indicate that you must choose one of the parameters within the braces; you do not 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:

cl430 −−run_linker {**−c** | **−cr**} *filenames* [−o *name.out*] **−l** *libraryname*

Related Documentation From Texas Instruments

To obtain a copy of any TI documents, call the Texas Instruments Literature Response Center at (800) 477−8924. When ordering, please identify the book by its title and literature number.

MSP430 Assembly Language Tools User's Guide (literature number SLAU131) describes the assembly language tools (the assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the MSP430 devices.

MSP430FE42x ESP430CE1 Peripheral Module User's Guide (literature number SLAU134) describes common peripherals available on the MSP430FE42x and ESP430CE1 ultra-low power microcontrollers. This book includes information on the setup, operation and registers of ESP430CE1

MSP430x1xx Family User's Guide (literature number SLAU049) describes the MSP430x1xx™ CPU architecture, instruction set, pipeline, and interrupts for these ultra-low power microcontrollers.

MSP430x3xx Family User's Guide (literature number SLAU012) describes the MSP430x3xx™ CPU architecture, instruction set, pipeline, and interrupts for these ultra-low power microcontrollers.

MSP430x4xx Family User's Guide (literature number SLAU056) describes the MSP430x4xx™ CPU architecture, instruction set, pipeline, and interrupts for these ultra-low power microcontrollers.

Related Documentation

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

ISO/IEC 9899:1999, International Standard − Programming Languages − C (The C Standard), International Organization for Standardization

ISO/IEC 9899:1989, International Standard − Programming Languages − C (The 1989 C Standard), International Organization for Standardization

ISO/IEC 14882−1998, International Standard − Programming Languages − C++ (The C++ Standard), International Organization for Standardization

ANSI X3.159−1989, Programming Language − C (Alternate version of the 1989 C Standard), 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.

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), by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

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

Trademarks

Contents

Figures

Tables

Examples

Notes

Chapter 1

Introduction

The MSP430 is supported by a set of software development tools, which includes an optimizing C/C++ compiler, an assembler, a linker, and assorted utilities.

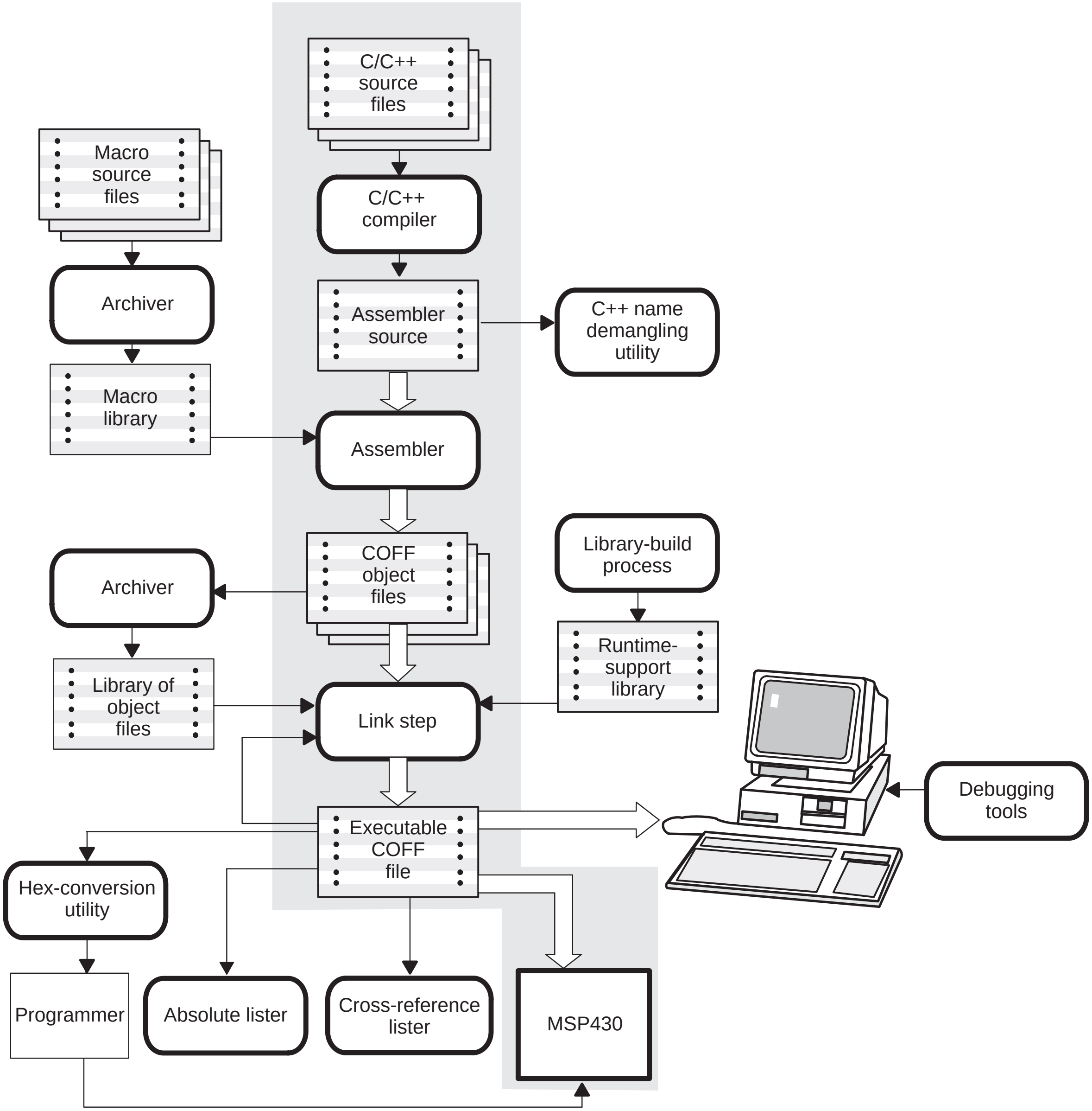
This chapter provides an overview of these tools and introduces the features of the optimizing C/C++ compiler. The assembler and linker are discussed in detail in the *MSP430 Assembly Language Tools User's Guide*.

Topic **Page**

1.1 Software Development Tools Overview

Figure 1−1 illustrates the MSP430 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. MSP430 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 MSP430 assembly language source code. An **optimizer** is part of the compiler. The optimizer modifies code to improve the efficiency of C/C++ programs.

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

- The **assembler** translates assembly language source files into machine language object files. The machine language is based on common object file format (COFF). The *MSP430 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/C++ Code*, for information about invoking the linker. See the *MSP430 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 *MSP430 Assembly Language Tools User's Guide* explains how to use the archiver.

- You can use the self-contained **library-build process**, rtssrc.zip, to build your own customized run-time-support library. Several standard libraries are included. See Chapter 8, *Library-Build Process*, for details.

  The **run-time-support libraries** contain the ANSI/ISO standard run-time-support functions, compiler-utility functions, and floating-point arithmetic functions that are supported by the MSP430 compiler. See Chapter 7, *Run-Time-Support Functions*, for more information.

- The MSP430 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, TI-TXT, ASCII-hex, Intel, Motorola-S, or Tektronix object format. The converted file can be downloaded to an EPROM programmer. The *MSP430 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 *MSP430 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 *MSP430 Assembly Language Tools User's Guide* explains how to use the cross-reference lister.

- 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. For more information, see Chapter 9, *C++ Name Demangler*.

- The main product of this development process is a module that can be executed in a MSP430 device. The code building and debugging tools are accessed within Code Composer Essentials. For more information, see the *Code Composer Essentials for MSP430* online help.

1.2 C/C++ Compiler Overview

The MSP430 C/C++ compiler is a full-featured optimizing compiler that translates standard ANSI/ISO C/C++ programs into MSP430 assembly language source. The following subsections describe the key features of the compiler.

1.2.1 ANSI/ISO Standard

The following features pertain to ANSI/ISO standards:

- **ANSI/ISO-standard C**

  The MSP430 C/C++ compiler conforms to the ANSI/ISO C standard as defined by the ISO specification and described in the second edition of Kernighan and Ritchie's *The C Programming Language* (K&R). The ISO C standard supercedes and is the same as the ANSI C standard.

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.

- **ANSI/ISO-standard C++**

  The MSP430 C/C++ compiler supports C++ as defined by the ISO C++ Standard and described in Ellis and Stroustrup's *The Annotated C++ Reference Manual* (ARM). The compiler also supports embedded C++.

  For a description of unsupported C++ features, see section 5.2, *Characteristics of TMS320C55x C++*, on page 5-5.

- **ANSI/ISO-standard run-time support**

  The compiler tools come with a complete run-time library. All library functions conform to the ANSI/ISO 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/ISO C subset as well as those components necessary for language support.

  For more information, see Chapter 7, *Run-Time-Support Functions.*

1.2.2 Output Files

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

- **Assembly source output**

  The compiler generates assembly language source files that you can inspect 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.

- **EPROM programmer data files**

  For stand-alone embedded applications, the compiler has the ability to place all code and initialization data into ROM, allowing C/C++ code to run from reset. The COFF files output by the compiler can be converted to EPROM programmer data files by using the hex conversion utility. For more information about the hex conversion utility, see the *MSP430 Assembly Language Tools User's Guide*.

1.2.3 Compiler Interface

The following features pertain to the compiler interface:

- **Compiler program**

  The compiler tools allow you to compile, assemble, and link programs in a single step. For more information, see section 2.2, *Invoking the C/C++ Compiler*, on page 2-4.

- **Flexible assembly language interface**

  The compiler has straightforward calling conventions, so you can write assembly and C/C++ functions that call each other. For more information, see Chapter 6, *Run-Time 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. Stand-alone 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 MSP430-specific optimizations take advantage of the features specific to the MSP430 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:

- The **library-build process** lets you custom-build object libraries from source for any combination of run-time models. For more information, see Chapter 8, *Library-Build Utility*.

- The **C++ name demangler utility** is a debugging aid that translates each mangled name it detects to its original name found in the C++ source code. For more information, see Chapter 9, *C++ Name Demangler*.

1.3 The Compiler and Code Composer Essentials

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

A Code Composer Essentials 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 Essentials, 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 Essentials's Project Properties→C/C++Build 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 Essentials and for information on setting code generation tool options within Code Composer Essentials, see the *Code Composer Essentials Online Help*.

Chapter 2

Using the C/C++ Compiler

The compiler translates your source program into code that the MSP430™ can execute. The source code must be compiled, assembled, and linked to create an executable object file. All of these steps are performed at once by using the compiler, cl430. This chapter provides a complete description of how to use cl430 to compile, assemble, and link your programs.

This chapter also describes the preprocessor, optimizer, inline function expansion features, and interlisting.

Topic		Page

2.1 About the Compiler

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

- The **compiler**, which includes the parser and optimizer, accepts C/C++ source code and produces MSP430 assembly language source code.

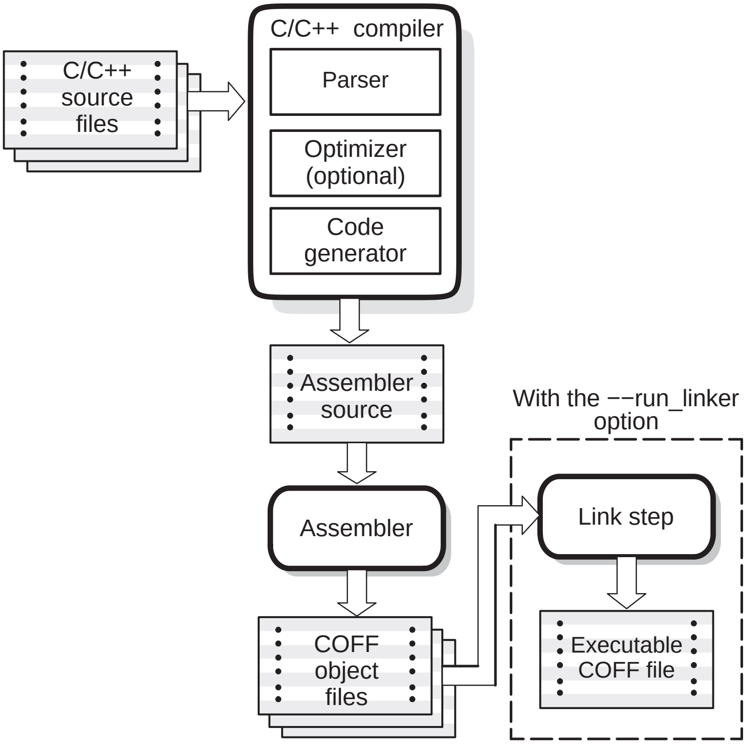
  You can compile C and C++ files in a single command—the compiler uses filename extensions to distinguish between them (see section 2.3.4, *Specifying Filenames*, for more information).

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

- The **link step** combines your object files to create an executable object file. The link step is optional, so you can compile and assemble many modules independently and link them later. See Chapter 4, *Linking C/C++ Code,* for information about linking files.

By default, the compiler does not perform the link step. You can invoke the link step by using the −−run_linker compiler option. Figure 2−1 illustrates the path the compiler takes with and without using the link step.

Figure 2−1. The Compiler Overview



For a complete description of the assembler and the link step, see the *MSP430 Assembly Language Tools User's Guide*.

2.2 Invoking the C/C++ Compiler

To invoke the compiler compiler, enter:

cl430 [*options*] [*filenames*] [**−−run_linker** [*link_options*] [*object files*]]

cl430	Command that runs the compiler and the assembler
options	Options that affect the way the input files are processed (the options are listed in Table 2−1 on page 2-6)
filenames	One or more C/C++ source files, assembly source files, or object files
−−run_linker	Option that invokes the link step. See Chapter 4, *Linking C/C++ Code,* for more information about invoking the link step.
link_options	Options that control the linking process
object files	Name of the additional object files for the linking process

The arguments to cl430 are of three types: compiler options, link step options, and files. The −−run_linker option is the signal that linking is to be performed. If the −−run_linker option is used, compiler options must precede the −−run_linker option, and other link step options must follow the −−run_linker option. Source code filenames must be placed before the −−run_linker option. Additional object file filenames may be placed after the −−run_linker option.

As an example, if you want to compile two files named symtab.c and file.c, assemble a third file named seek.asm, and link to create an executable file, you enter:

```
cl430 symtab.c file.c seek.asm −run_linker −−library=lnk.cmd
    −−library=rts430.lib
```

Entering this command produces the following output:

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

2.3 Changing the Compiler’s Behavior With Options

Options control the operation of the compiler. 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. For the most current summary of the options, enter cl430 with no parameters on the command line.

The following information applies to the compiler options:

- Options are preceded by one or two hyphens.
- Options are case sensitive.
- Options are sequences of characters.
- Individual options cannot be combined.
- An option with a *required* parameter should be specified with an equal sign before the parameter to clearly associate the parameter with the option. For example, the option to undefine a name can be expressed as −−undefine=*name*. Although not recommended, you can separate the option and the parameter without a space, as in −−undefine*name*.
- An option with an *optional* parameter should be specified with an equal sign before the parameter to clearly associate the parameter with the option. For example, the option to specify the maximum amount of optimization can be expressed as −−opt_level=3. Although not recommended, you can specify the parameter directly after the option, as in −−opt_level3. No space is allowed between the option and the optional parameter, so −O 3 is not accepted.
- Files and options except the −−run_linker option can occur in any order. The −−run_linker option must follow all other compiler options and precede any link step options.

You can define default options for the compiler by using the MSP430_C_OPTION environment variable. For more information on the environment variable, see section 2.4.2, *Setting Default Compiler Options (MSP430_C_OPTION)*, on page 2-23.

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

For an online summary of the options, enter **cl430** with no parameters on the command line.

Table 2−1. Compiler Options Summary

(a) Options that control the compiler

Option	Alias	Effect	Page
−−c_src_interlist	−ss	Interlists C source and assembly statements	2-14, 3-13
−−cmd_file=*filename*	−@	Interprets contents of *filename* as an extension to the command line	2-14
−−compile_only	−c	Disables linking (negates −−run_linker)	2-14, 4-4
−−compiler_revision		Prints out the compiler release revision and exits.	−−
−−define=*name* [=*def*]	−D	Predefines *name*	2-14
−−gen_aux_user_info	−b	Generates auxiliary information file	2-14
−−help	−h	Show help screen.	2-14
−−include_path=*directory*	−I	Defines #include search path	2-14, 2-26
−−keep_asm	−k	Keeps the assembly language (.asm) file	2-15
−−quiet	−q	Suppresses common compiler non-diagnostic output.	2-15
−−run_linker	−z	Enables linking	2-15
−−skip_assembler	−n	Compiles only	2-10
−−src_interlist	−s	Interlists optimizer comments (if available) and assembly statements; otherwise, interlists C source and assembly source statements	2-15, 2-39
−−super_quiet	−qq	Suppresses all compiler non-diagnostic output.	2-15
−−undefine=*name*	−U	Undefines *name*	2-15
−−verbose		Displays banner and function progress information.	2-15
−−tool_version	−version	Displays version number for each tool	−−

Table 2−1. Compiler Options Summary (Continued)

(b) Options that control symbolic debugging and profiling

Option	Alias	Effect	Page
−−symdebug:dwarf	−g	Enables symbolic debugging using the DWARF debugging format	2-17
−−symdebug:none		Disables all symbolic debugging	2-17
−−symdebug:skeletal		Enables minimal symbolic debugging that does not hinder optimizations (default behavior)	2-17

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

Option	Alias	Effect	Page
−−asm_extension=.*extension*	−ea	Sets default extension for assembly files	2-20
−−obj_extension=.*extension*	−eo	Sets default extension for object files	2-20
−−c_extension=.*extension*	−ec	Sets default extension for C source files	2-20
−−cpp_extension=.*extension*	−ep	Sets default extension for C++ source files	2-20
−−listing_extension=.*extension*	−es	Sets default extension for assembly listing files	2-20

(d) Options that specify files

Option	Alias	Effect	Page
−−asm_file=*filename*	−fa	Identifies *filename* as an assembly file, regardless of its extension. By default, the compiler treats .asm or .s* (extension begins with s) as an assembly file.	2-19
−−c_file=*filename*	−fc	Identifies *filename* as a C file, regardless of its extension. By default, the compiler treats .c or no extension as a C file.	2-19
−−cpp_file=*filename*	−fp	Identifies *filename* as a C++ file, regardless of its extension. By default, the compiler treats .C, .cpp, .cc, or .cxx as a C++ file.†	2-19
−−cpp_default	−fg	Causes all C files to be treated as C++ files.	2-19
−−obj_file=*filename*	−fo	Identifies *filename* as an object file, regardless of its extension. By default, the compiler treats .o* as an object file.	2-19

Table 2−1. Compiler Options Summary (Continued)

(e) Options that specify directories

Option	Alias	Effect	Page
−−abs_directory=*directory*	−fb	Specifies absolute listing file directory	2-21
−−asm_directory=*directory*	−fs	Specifies assembly file directory	2-21
−−list_directory=*directory*	−ff	Specifies listing/xref file directory	2-21
−−obj_directory=*directory*	−fr	Specifies object file directory	2-21
−−temp_directory=*directory*	−ft	Specifies temporary file directory	2-21

(f) Options that control parsing

Option	Alias	Effect	Page
−−embedded_cpp	−pe	Enables embedded C++ mode	5-27
−−exceptions		Enable C++ exception handling	2-14
−−gen_acp_raw	−pl	Outputs raw listing information	2-35
−−gen_acp_xref	−px	Generates a cross-reference listing file	2-34
−−kr_compatible	−pk	Allows K&R compatibility	5-25
−−multibyte_chars	−pc	Multibyte character support	−−
−−no_inlining	−pi	Disables definition-controlled inlining (but −O3 optimizations still perform automatic inlining)	2-38
−−no_intrinsics	−pn	Disables intrinsic functions	−−
−−program_level_compile	−pm	Combines source files to perform program-level optimization	3-6
−−relaxed_ansi	−pr	Enables relaxed mode; ignores strict ANSI violations	5-27
−−rtti	−rtti	Enables run-time type information (RTTI), which allows the type of an object to be determined at run time.	5-5
−−strict_ansi	−ps	Enables strict ANSI mode (for C/C++, not K&R C)	5-27

Table 2−1. Compiler Options Summary (Continued)

(g) Parser options that control preprocessing

Option	Alias	Effect	Page
−−preproc_dependency	−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
−−preproc_includes[**=***file*]	−ppi	Performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive	2-28
−−preproc_only	−ppo	Performs preprocessing only. Writes preprocessed output to a file with the same name as the input but with a .pp extension	2-27
−−preproc_with_comment	−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	2-28
−−preproc_with_compile	−ppa	Continues compilation after preprocessing	2-27
−−preproc_with_line	−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	2-28

Table 2−1. Compiler Options Summary (Continued)

(h) Parser options that control diagnostics

Option	Alias	Effect	Page
−−diag_suppress=*num*	−pds	Suppresses the diagnostic identified by *num*	2-31
−−diag_error=*num*	−pdse	Categorizes the diagnostic identified by *num* as an error	2-31
−−diag_remark=*num*	−pdsr	Categorizes the diagnostic identified by *num* as a remark	2-31
−−diag_warning=*num*	−pdsw	Categorizes the diagnostic identified by *num* as a warning	2-29
−−display_error_number=*num*	−pden	Displays a diagnostic's identifiers along with its text	2-31
−−issue_remarks	−pdr	Issues remarks (nonserious warnings)	2-31
−−no_warnings	−pdw	Suppresses warning diagnostics (errors are still issued)	2-31
−−set_error_limit=*num*	−pdel	Sets the error limit to *num*. The compiler abandons compiling after this number of errors. (The default is 100.)	2-32
−−verbose_diagnostics	−pdv	Provides verbose diagnostics that display the original source with line-wrap	2-32
−−write_diagnostics_file	−pdf	Generates a diagnostics information file	2-31

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

Option	Alias	Effect	Page
−−aliased_variables	−ma	Assumes variables are aliased	2-16
−−plain_char={signed|unsigned}	−mc	Changes variables of type char from unsigned to signed	2-16
−−gen_func_subsections		Turns on function subsections	4-14
−−opt_for_space	−ms	Optimizes for code size (default)	−−
−−opt_for_speed	−mf	Optimizes for speed over space	2-16
−−optimize_with_debug	−mn	Enables optimizations disabled by −g	2-16
−−printf_support={full|minimal|nofloat}	−−	Enables support for smaller limited versions of printf.	2-16
−−small−enum		Uses the smallest possible size for the enumeration type	2-17
−−silicon_version={msp|mspx}	−v	Selects instruction set.	2-17

Table 2−1. Compiler Options Summary (Continued)

(j) Options that control optimization

Option	Alias	Effect	Page
−−auto_inline=*size*	−oi	Sets automatic inlining size (−O3 only)	3-12
−−call_assumptions=0	−op0	Specifies that the module contains functions and variables that are called or modified from outside the source code provided to the compiler	3-7
−−call_assumptions=1	−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	3-7
−−call_assumptions=2	−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)	3-7
−−call_assumptions=3	−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	3-7
−−gen_opt_info=0	−on0	Disables optimizer information file	3-5
−−gen_opt_info=1	−on1	Produces optimizer information file	3-5
−−gen_opt_info=2	−on2	Produces verbose optimizer information file	3-5
−−opt_level=0	−O0	Optimizes register usage	3-2
−−opt_level=1	−O1	Uses −O0 optimizations and optimizes locally	3-2
−−opt_level=2	−O2 or −O	Uses −O1 optimizations and optimizes globally	3-3
−−opt_level=3	−O3	Uses −O2 optimizations and optimizes file	3-3, 3-4
−−optimizer_interlist	−os	Interlists optimizer comments with assembly statements	3-13
−−std_lib_func_redefined	−ol0 (−oL0)	Informs the optimizer that your file alters a standard library function	3-4
−−std_lib_func_defined	−ol1 (−oL1)	Informs the optimizer that your file declares a standard library function	3-4
−−std_lib_func_not_defined	−ol2 (−oL2)	Informs the optimizer that your file does not declare or alter library functions. Overrides the standard library function options: −−std_lib_func_redefined and −−std_lib_func_defined.	3-4

Table 2−1. Compiler Options Summary (Continued)

(k) Options that control the assembler

Option	**Alias**	**Effect**	**Page**
−−absolute_listing	−aa	Enables absolute listing	2-22
−−asm_define=*name*[=*value*]	−ad	Sets the *name* symbol	2-22
−−asm_dependency	−apd	Performs preprocessing; lists only assembly dependencies	2-22
−−asm_includes	−api	Performs preprocessing; lists only included #include files	2-22
−−asm_listing	−al	Produces a listing file	2-22
−−asm_undefine=*name*	−au	Undefines the predefined constant *name*	2-22
−−copy_file=*filename*	−ahc	Copies the specified file for the assembly module	2-22
−−cross_reference	−ax	Generates the cross-reference file	2-22
−−include_file=*filename*	−ahi	Includes the file for the assembly module	2-22
−−output_all_syms	−as	Puts labels in the symbol table	2-22
−−syms_ignore_case	−ac	Makes case significant in assembly source files	2-22

(l) Options that control the link step

Options	**Alias**	**Effect**	**Page**
−−absolute_exe	−a	Generates absolute output	4-5
−ar		Generates relocatable output	4-5
−−arg_size=*size*	−−args	Allocates memory to be used by the loader to pass arguments	4-5
−−disable_clink	−j	Disables conditional linking	4-5
−−entry_point=*global_symbol*	−e	Defines entry point	4-5
−−fill_value=*value*	−f	Defines fill value	4-5
−−heap_size=*size*	−heap −−heap	Sets heap size (in bytes)	4-5
−−ignore_align_flags	−k	Ignores alignment flags in input sections	4-5
−−library=*filename*	−l	Supplies library name	4-5
−−linker_help	−help −−help	Displays usage information	4-5
−−make_global=*symbol*	−g	Keeps a *symbol* global (overrides −h)	4-6

Table 2−1. Compiler Options Summary (Continued)

(l) Options that control the link step

Options	**Alias**	**Effect**	**Page**
−−make_static	−h	Makes global symbols static	4-6
−−map_file=*filename*	−m	Names the map file	4-6
−−no_sym_merge	−b	Disables merge of symbolic debugging information	4-6
−−no_symtable	−s	Strips symbol table	4-6
−−output_file=*filename*	−o	Names the output file	4-6
−−priority	−priority	Searches libraries in order specified	4-6
−−ram_model	−cr	Autoinitializes variables at load time	4-6
−−relocatable	−r	Retains relocation entries in the output module	4-6
−−reread_libs	−x	Forces rereading of libraries	4-6
−−rom_model	−c	Autoinitializes variables at run time	4-6
−−run_abs	−abs	Produces an absolute listing file	4-6
−−search_path=*directory*	−I	Defines library search path	4-6
−−stack_size=*size*	−stack	Sets stack size (in bytes)	4-6
−−undef_sym=*symbol*	−U	Undefines *symbol*	4-6
−−use_hw_mpy		Replaces default multiply routine with hardware multiplier routine	4-7
−−warn_sections	−w	Warns if an unspecified output section is created	4-7
−−xml_link_info		Generates an XML information file	4-7

2.3.1 Frequently Used Generic Options

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

−−c_src_interlist Invokes the interlist feature, which interweaves original C/C++ source with compiler-generated assembly language. If the optimizer is invoked (−−opt_level=*n* option) along with this option, your code might be reorganized substantially. For more information, see section 2.10, *Using Interlist,* on page 2-39.

−−cmd_file=*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"

−−compile_only Suppresses the link step and overrides the −−run_linker option, which specifies linking. Use this option when you have −−run_linker specified in the MSP430_C_OPTION environment variable and you do not want to link. For more information, see section 4.1.3, *Disabling the Link Step (−−compile_only Compiler Option)*, on page 4-4.

−−define=*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.

−−exceptions Enables C++ exception handling. The compiler will generate code to handle try/catch/throw statements in C++ code. See Section 5.5, *C++ Exception Handling*, on page 5-10.

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

−−help Shows usage information. When −−help is followed by another option, detailed information about that option is displayed. When −−help is followed by a phrase, options related to the phrase are searched for and displayed.

−−include_path=*directory* Adds *directory* to the list of directories that the compiler searches for #include files. You can use this option several times to define several directories; be sure to separate −−include_path options with spaces. If you do not specify a directory name, the preprocessor ignores the −−include_path option. For more information, see section 2.5.2.1, *Changing the #include File Search Path With the −−include_path Option*, on page 2-26.

−−keep_asm Keeps the assembly language output from the compiler. Normally, the compiler deletes the output assembly language file after assembly is complete.

−−quiet Suppresses common compiler non-diagnostic output including banners and progress information from all the tools. Remarks, errors, and warnings are still generated, as well as feature-specific information.

−run_abs Generates an absolute listing file when used after the −−run_linker option. You must use the −−output_file option (after −−run_linker) to specify the .out file for the absolute lister, even if you use a link step command file that already uses −−output_file.

−−run_linker Runs the link step on the specified object files. The −−run_linker option and its parameters follow all other options on the command line. All arguments that follow −−run_linker are passed to the link step. For more information, see section 4.1, *Invoking the Link Step (−−run_linker Option)*, on page 4-2.

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

−−src_interlist Invokes the interlist feature, which interweaves optimizer comments *or* C/C++ source with assembly source. If the optimizer is invoked (−−opt_level=*n* option), optimizer comments are interlisted with the assembly language output of the compiler, which may rearrange code substantially. If the optimizer is not invoked, C/C++ source statements are interlisted with the assembly language output of the compiler, allowing you to inspect the code generated for each C/C++ statement. The −−src_interlist option implies the −−keep_asm option. For more information about using the interlist feature with the optimizer, see section 3.7, *Using Interlist With Optimization*, on page 3-13.

−−super_quiet Suppresses all compiler non-diagnostic output including banners and progress information from all the tools. Only source filenames and error messages are output.

−−undefine=*name* Undefines the predefined constant *name*. The −−undefine option overrides any −−define options for the specified constant.

−−verbose Displays progress information and toolset version. Resets the −−quiet and −−super_quiet options.

2.3.2 MSP430-Specific Options

Here are detailed descriptions of the MSP430 device-specific options:

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

−−opt_for_speed Optimizes your code for speed over size. By default, the MSP430 optimizer attempts to reduce the size of your code at the expense of speed.

−−optimize_with_debug Re-enables the optimizations disabled by the −−symdebug:dwarf option. If you use the −−symdebug:dwarf option, many code generator optimizations are disabled because they disrupt the debugger. Therefore, if you use the −−optimize_with_debug option, portions of the debugger's functionality will be unreliable.

−−plain_char Changes C/C++ variables of type char from unsigned to signed. By default, char variables are unsigned.

−−printf_support=*version* Enables support for smaller, limited versions of the printf and sprintf run-time-support functions. The valid *version* values are full (default), nofloat and minimal.

The printf/sprintf functions use a common low-level routine, _printfi, which processes a given printf format string. The full version of_printfi provides formatting capabilities that are not required in typical embedded applications. To address this the C run-time libraries also include two limited of versions of _printfi that provide a useful subset of the formatting features specified by C library standard.

One version excludes support for printing floating values. All format specifiers except for %f, %g, %G, %e, or %E are supported. You specify this version with −−printf_support=nofloat when compiling and linking.

The other version only supports printing of integer, char, or string values without width or precision flags. Specifically, only the %%, %d, %o, %c, %s, and %x format specifiers are supported. You specify this version with −−printf_support=minimal when compiling and linking.

There are no run-time error checks if a format specifier is used but is not supported in the version specified by the −−printf_support option. An upcoming release will add compile-time checks.

The −−printf_support option precedes the −−run_linker option; −−printf_support must be used when performing the final link.

−−small−enum By default, the MSP430 compiler uses 16 bits for every *enum*. When you use the −−small−enum option, the smallest possible byte size for the enumeration type is used. For example, enum example_enum {first = −128, second = 0, third = 127} will use only one byte instead of 16 bits when the −−small−enum option is used.

Do not link object files compiled with the −−small−enum option with object files that have been compiled without it. If you use the −−small−enum option, you must use it with all of your C/C++ files, otherwise, you will encounter errors that cannot be detected until run time.

−−silicon_version Selects the intruction set version. Using −−silicon_version=mspx generates code for MSP430x devices (20-bit code addresses). Using −−silicon_version=msp generates code for 16-bit MSP430 devices.

Modules assembled/compiled for 16-bit MSP devices are not compatible with modules that are assembled/compiled for 20-bit MSPx devices. The link step generates errors if an attempt is made to combine incompatible object files.

2.3.3 Symbolic Debugging and Profiling Options

The following options are used to select symbolic debugging or profiling:

−g or **−−symdebug:dwarf** Generates directives that are used by the C/C++ source-level debugger and enables assembly source debugging in the assembler. The −g option disables many code generator optimizations, because they disrupt the debugger. You can use the −g option with the −−opt_level option (−O) to maximize the amount of optimization that is compatible with debugging (see section 3.8, *Debugging Optimized Code*, on page 3-15.

For more information on the DWARF debug format, see the *DWARF Debugging Information Format Specification*, 1992−1993, UNIX International, Inc.

−−symdebug:none Disables all symbolic debugging output. This option is not recommended; it prevents debugging and most performance analysis capabilities.

−−symdebug:skeletal Generates symbolic debugging for program analysis including as much information as possible without hindering optimization. Generally, this consists of global-scope information only. This option reflects the default behavior of the compiler.

2.3.4 Specifying Filenames

The input files that you specify on the command line can be C++ source files, C source files, assembly source files, or object files. The compiler 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.

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

For information about how you can alter the way the compiler interprets individual filenames, see section 2.3.5. For information about how you can alter the way the compiler interprets and names the extensions of assembly source and object files, see section 2.3.7.

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:

```
cl430 *.c
```

2.3.5 Changing How the Compiler Interprets Filenames

You can use options to change how the compiler interprets your filenames. If the extensions that you use are different from those recognized by the compiler, you can use the −−asm_file, −−c_file, −−obj_file, and −−cpp_file 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:

−−asm_file=*filename*	for an assembly language source file
−−c_file=*filename*	for a C source file
−−cpp_file=*filename*	for a C++ source file
−−obj_file=*filename*	for an object file

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

```
cl430 --cpp_file=file.s --asm_file=assy
```

You cannot use these options with wildcard specifications.

2.3.6 Changing How the Compiler Processes C Files

The −−cpp_default 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 2.3.4 on page 2-18 for more information about filename extension conventions.

2.3.7 Changing How the Compiler Interprets and Names Extensions

You can use options to change how the compiler interprets filename extensions and names the extensions of the files that it creates. On the command line, these 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:

−−asm_extension=**.***extension*	for an assembly language file
−−c_extension=**.***extension*	for a C source file
−−cpp_extension=**.***extension*	for a C++ source file
−−listing_extension=**.***extension*	for an assembly listing file
−−obj_extension=**.***extension*	for an object 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:

```
cl430 --asm_extension=.rrr --obj_extension=.o fit.rrr
```

The period (.) in the extension is optional. The preceding example could be written as:

```
cl430 --asm_extension=rrr --obj_extension=o fit.rrr
```

2.3.8 Specifying Directories

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

−−abs_directory=*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 −−abs_directory option:

```
cl430 --abs_directory=d:\object ...
```

−−asm_directory=*directory* Specifies a directory for assembly files. To specify an assembly file directory, type the directory’s pathname on the command line after the −−asm_directory option:

```
cl430 --asm_directory=d:\assembly ...
```

−−list_directory=*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 (−−asm_listing) option or cross-reference listing (−−cross_reference) option causes the compiler to act as if the −−asm_listing option was specified. To specify a listing file directory, type the directory’s pathname on the command line after the −−listing_directory option:

```
cl430 --list_directory=d:\object ...
```

−−obj_directory=*directory* Specifies a directory for object files. To specify an object file directory, type the directory’s pathname on the command line after the −−obj_directory option:

```
cl430 --obj_directory=d:\object ...
```

−−temp_directory=*directory* Specifies a directory for temporary intermediate files. To specify a temporary directory, insert the directory’s pathname on the command line after the −−temp_directory option:

```
cl430 --temp_directory=d:\temp ...
```

2.3.9 Options That Control the Assembler

These are the assembler options that you can use with the compiler to invoke and control the assembler:

−−absolute_listing Creates an absolute listing. An absolute listing shows the absolute addresses of the object code.

−−asm_define=*name* **−−asm_define**=*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.

−−asm_dependency Performs preprocessing for assembly files, 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 .ppa extension.

−−asm_includes Performs preprocessing for assembly files, 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 .ppa extension.

−−asm_listing Produces an assembly listing file.

−−asm_undefine=*name* Undefines the predefined constant *name*, which overrides any −−asm_define options for the specified constant.

−−copy_file=*filename* Copies the specified file for the assembly module. The file is inserted before source file statements. The copied file appears in the assembly listing files.

−−cross_reference Produces a symbolic cross-reference in the listing file.

−−include_file=*filename* Includes 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.

−−output_all_syms Puts labels in the symbol table. Label definitions are written to the COFF symbol table for use with symbolic debugging.

−−syms_ignore_case Makes case insignificant in the assembly language source files. For example, −−syms_ignore_case makes the symbols ABC and abc equivalent. If you do not use this option, case is significant (this is the default).

For more information about assembler options, see the *MSP430 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.

Note: C_DIR and C_OPTION

The C_DIR and C_OPTION environment variables are deprecated. Use the device-specific environment variables instead.

2.4.1 Specifying Directories (MSP430_C_DIR)

The compiler uses the MSP430_C_DIR environment variable to name alternate directories that contain #include files. To specify directories for #include files, set MSP430_C_DIR with this Windows™ command syntax:

set MSP430_C_DIR=*directory1*[**;***directory2* ...]

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

2.4.2 Setting Default Compiler Options (MSP430_C_OPTION)

You might find it useful to set the compiler, assembler, and link step default options using the MSP430_C_OPTION environment variable. If you do this, the compiler uses the default options and/or input filenames that you name with this variable every time you run the compiler.

Setting the default options with these environment variables is useful when you want to run the compiler consecutive times with the same set of options and/or input files. After the compiler reads the command line and the input filenames, it looks for the MSP430_C_OPTION environment variable and processes it.

Set the environment variables with this Windows syntax:

MSP430_C_OPTION="*option*$_1$ [*option*$_2$ . . .]**"; export MSP430_C_OPTION**

Environment variable options are specified in the same way and have the same meaning as they do on the command line. For example, if you want to always run quietly (the −−quiet option), enable C/C++ source interlisting (the −−src_interlist option), and link (the −−run_linker option), set up the MSP430_C_OPTION environment variable as follows:

```
set MSP430_C_OPTION=--quiet --src_interlist --run_linker
```

In the following examples, each time you run the compiler, it runs the link step. Any options following −−run_linker on the command line or in MSP430_C_OPTION are passed to the link step. This enables you to use the MSP430_C_OPTION environment variable to specify default compiler and link step options and then specify additional compiler and link step options on the command line. If you have set −−run_linker in the environment variable and want to compile only, use the −−compile_only option of the compiler. These additional examples assume MSP430_C_OPTION is set as shown above:

```
cl430 *.c                        ; compiles and links
cl430 --compile_only *.c         ; only compiles
cl430 *.c --run_linker lnk.cmd   ; compiles/links using .cmd file
cl430 --compile_only *.c --run_linker lnk.cmd ; only compiles
                        ; (--compile_only overrides --run_linker)
```

For more information about compiler options, see section 2.3, *Changing the Compiler's Behavior With Options*, on page 2-5. For more information about link step options, see section 4.2, *Link Step Options*, on page 4-5.

2.5 Controlling the Preprocessor

This section describes specific features of the preprocessor. The MSP430 C/C++ compiler includes standard C preprocessing functions, which are built into the first pass of the compiler. The preprocessor handles:

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

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

2.5.1 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
_ _MSP430_ _	Always defined
_ _MSP430X461X_ _ _ _msp430x461x_ _	Defined if −−silicon_version=mspx is specified
_ _unsigned_chars_ _	Defined if char types are unsigned by default (default or −−plain_char=unsigned)
_ _signed_chars_ _	Defined if char types are signed by default (−−plain_char=signed)
_ _DATE_ _†	Expands to the compilation date in the form *mm dd yyyy*
_ _FILE_ _†	Expands to the current source filename
_ _LARGE_CODE_MODEL_ _	Defined if −−silicon_version=mspx is specified
_ _LINE_ _†	Expands to the current line number
_ _TIME_ _†	Expands to the compilation time in the form *hh:mm:ss*
_ _TI_COMPILER_VERSION_ _	Expands to an integer value representing the current compiler version number. For example, version 1.20 is represented as 0012000.
_INLINE	Expands to 1 if optimization is used; undefined otherwise. Regardless of any optimization, always undefined when −pi is used.

† Specified by the ANSI/ISO 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 2006”);
```

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 −−include_path option
  3) Directories set with the MSP430_C_DIR environment variable
- If you enclose the filename in angle brackets (< >), the compiler searches for the file in the following directories in this order:
  1) Directories named with the −−include_path option
  2) Directories set with the MSP430_C_DIR environment variable

See section 2.5.2.1, *Changing the #include File Search Path With the −−include_path Option*, for information on using the −−include_path option. See section 2.4.1, *Specifying Directories (MSP430_C_DIR)*, on page 2-23, for information on using the MSP430_C_DIR environment variable.

2.5.2.1 *Changing the #include File Search Path With the −−include_path Option*

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

−−include_path=*directory1* [**−−include_path**=*directory2* ...]

Each −−include_path 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 −−include_path 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:\MSP430_C_DIRtools\files\alt.h
UNIX	/MSP430_C_DIRtools/files/alt.h

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

Operating System	Enter
Windows	`cl430 --include_path=c:\MSP430tools\files source.c`
UNIX	`cl430 --include_path/MSP430tools/files source.c`

2.5.3 Generating a Preprocessed Listing File (−−preproc_only Option)

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

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

2.5.4 Continuing Compilation After Preprocessing (−−preproc_with_compile 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 −−preproc_with_compile option along with the other preprocessing options. For example, use −−preproc_with_compile with −−preproc_only 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 (−−preproc_with_comment Option)

The −−preproc_with_comment 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 −−preproc_with_comment option instead of the −−preproc_only option if you want to keep the comments.

2.5.6 Generating a Preprocessed Listing File With Line-Control Information (−−preproc_with_line Option)

By default, the preprocessed output file contains no preprocessor directives. If you want to include the #line directives, use the −−preproc_with_line option. The −−preproc_with_line 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 (−−preproc_dependency Option)

The −−preproc_dependency 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. Optionally, you can specify a filename for the output, for example:

```
cl55 --preproc_dependency=make.pp file.c
```

2.5.8 Generating a List of Files Included With the #include Directive (−−preproc_includes Option)

The −−preproc_includes 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. Optionally, you can specify a filename for the output, for example:

```
cl55 --preproc_includes=include.pp file.c
```

2.6 Understanding Diagnostic Messages

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

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

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

Diagnostic messages have an associated severity, as follows:

- A **fatal error** indicates a problem of such severity that the compilation cannot continue. Examples of problems that can cause a fatal error include command-line errors, internal errors, and missing include files. If multiple source files are being compiled, any source files after the current one will not be compiled.
- An **error** indicates a violation of the syntax or semantic rules of the C or C++ language. Compilation continues, but object code is not generated.
- A **warning** indicates something that is valid but questionable. Compilation continues and object code is generated (if no errors are detected).
- A **remark** is less serious than a warning. It indicates something that is valid and probably intended, but may need to be checked. Compilation continues and object code is generated (if no errors are detected). By default, remarks are not issued. Use the −−issue_remarks compiler 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 −−verbose_diagnostics compiler 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 (−−display_error_number) to request that the diagnostic's numeric identifier be included in the diagnostic message. When displayed, the diagnostic identifier also indicates whether the diagnostic can have its severity overridden on the command line. If the severity can be overridden, the diagnostic identifier includes the suffix −D (for *discretionary*); otherwise, no suffix is present. For example:

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

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

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

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

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

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

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

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

2.6.1 Controlling Diagnostics

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

−−diag_error= *num* Categorizes the diagnostic identified by *num* as an error. To determine the numeric identifier of a diagnostic message, use the −−display_error_number option first in a separate compile. Then use −−diag_error=*num* to recategorize the diagnostic as an error. You can alter the severity of discretionary diagnostics only.

−−diag_remark=*num* Categorizes the diagnostic identified by *num* as a remark. To determine the numeric identifier of a diagnostic message, use the −−display_error_number option first in a separate compile. Then use −−diag_remark=*num* to recategorize the diagnostic as a remark. You can alter the severity of discretionary diagnostics only.

−−diag_suppress=*num* Suppresses the diagnostic identified by *num*. To determine the numeric identifier of a diagnostic message, use the −−display_error_number option first in a separate compile. Then use −−diag_suppress=*num* to suppress the diagnostic. You can suppress only discretionary diagnostics.

−−diag_warning= *num* Categorizes the diagnostic identified by *num* as a warning. To determine the numeric identifier of a diagnostic message, use the −−display_error_number option first in a separate compile. Then use −−diag_warning=*num* to recategorize the diagnostic as a warning. You can alter the severity of discretionary diagnostics only.

−−display_error_number=*num* 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 (−−diag_suppress, −−diag_error, −−diag_remark, and −−diag_warning).

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.

−−issue_remarks Issues remarks (nonserious warnings), which are suppressed by default

−−no_warnings Suppresses warning diagnostics (errors are still issued)

−−set_error_limit=*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.)

−−verbose_diagnostics — Provides verbose diagnostics that display the original source with line-wrap and indicate the position of the error in the source line

−−write_diagnostics_file — Writes diagnostics to a file rather than standard error. The filename will be the same as the input file but with a .err extension.

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 −−quiet option, this is the result:

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

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

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

Next, you can use the diagnostic identifier of 111 as the argument to the −−diag_remark 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 one of the strings ">> WARNING:" or ">> ERROR:" preceding the message.

For example:

```
cl430 –j
>> WARNING: invalid option –j (ignored)
>> ERROR: no source files
```

2.7 Generating Cross-Reference Listing Information (−−gen_acp_xref *Option*)

The −−gen_acp_xref compiler option generates a cross-reference listing file (.crl) that contains reference information for each identifier in the source file. (The −−gen_acp_xref option is separate from −−cross_reference, which is an assembler rather than a compiler 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 (−−gen_acp_raw Option)

The −−gen_acp_raw 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 −−preproc_only, −−preproc_with_comment, and −−preproc_with_line 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 −−gen_acp_raw 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*, on page 2-29.

2.9 Using Inline Function Expansion

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

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

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

- Intrinsic operators are inlined by default.
- Code is compiled with definition-controlled inlining.
- When the optimizer is invoked with the −−opt_level=3 option (−O3), automatic inline expansion is performed at call sites to small functions. For more information about automatic inline function expansion, see section 3.6, *Automatic Inline Expansion (−−auto_inline*=size *Option)*, on page 3-12.

Note: Function Inlining Can Greatly Increase Code Size

Expanding functions inline expands code size, and inlining a function that is called a great number of times greatly increases code size. Function inlining is optimal for small functions that are called infrequently.

2.9.1 Inlining Intrinsic Operators

An operator is intrinsic if it can be implemented very efficiently with the target's instruction set. The compiler automatically inlines the intrinsic operators of the target system by default. Inlining happens whether or not you use the optimizer and whether or not you use any compiler or optimizer options on the command line. These functions are considered the intrinsic operators:

- abs
- labs
- fabs

2.9.2 Using the inline Keyword, the −−no_inlining Compiler Option, and −−opt_level=O3 Optimization

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

The inline keyword specifies that a function is expanded inline at the point at which it is called, rather than by using standard calling procedures.

The semantics of the inline keyword follows that described in the C++ working paper. The inline keyword is identically supported in C as a language extension. Because it is a language extension that could conflict with a strictly conforming program, however, the keyword is disabled in strict ANSI C mode (when you use the −−strict_ansi compiler option). If you want to use definition-controlled inlining while in strict ANSI C mode, use the alternate keyword _ _inline.

When you want to compile without definition-controlled inlining, use the −−no_inlining option.

Note: Using the −−no_inlining Option With −−opt_level=3 (−O3) Optimizations

When you use the −−no_inlining option with −−opt_level=3 optimizations, automatic inlining is still performed.

2.10 Using Interlist

The compiler tools allow you to interlist C/C++ source statements into the assembly language output of the compiler. Interlisting enables you to inspect the assembly code generated for each C/C++ statement. Interlisting 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 feature is to use the −−src_interlist option. To compile and run the interlist utility on a program called function.c, enter:

```
cl430 -s function.c
```

The −−src_interlist option prevents the compiler from deleting the interlisted assembly language output file. The output assembly file, function.asm, is assembled normally.

When you invoke interlisting without the optimizer, interlist 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 2−1 shows a typical interlisted assembly file. For more information about using interlist with the optimizer, see section 3.7, *Using Interlist With the Optimizer*, on page 3-13.

Example 2−1. An Interlisted Assembly Language File

```
;******************************************************************************
;* MSP430 C/C++ Codegen                                           Unix v0.2.0 *
;* Date/Time created: Tue Jun 29 14:54:28 2004                                *
;******************************************************************************
        .compiler_opts --mem_model:code=flat --mem_model:data=flat --symdebug:none
;       acp430 −@/var/tmp/TI764/AAAv0aGVG
        .sect   ".text"
        .align  2
        .clink
        .global main
;-----------------------------------------------------------------------
;   3 | int main()
;-----------------------------------------------------------------------

;******************************************************************************
;* FUNCTION NAME: main                                                        *
;*                                                                            *
;*   Regs Modified     : SP,SR,r11,r12,r13,r14,r15                            *
;*   Regs Used         : SP,SR,r11,r12,r13,r14,r15                            *
;*   Local Frame Size  : 2 Args + 0 Auto + 0 Save = 2 byte                    *
;******************************************************************************
main:
;* ---------------------------------------------------------------------------*
        SUB.W     #2,SP
;-----------------------------------------------------------------------
;   5 | printf("Hello, world\n");
;-----------------------------------------------------------------------
        MOV.W     #$C$SL1+0,0(SP)       ; |5|
        CALL      #printf               ; |5|
                                          ; |5|
;-----------------------------------------------------------------------
;   7 | return 0;
;-----------------------------------------------------------------------
        MOV.W     #0,r12                ; |7|
        ADD.W     #2,SP                 ; |7|
        RET       ; |7|
        ; |7|
;******************************************************************************
;* STRINGS                                                                    *
;******************************************************************************
        .sect   ".const"
        .align  2
$C$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 debug optimized code.

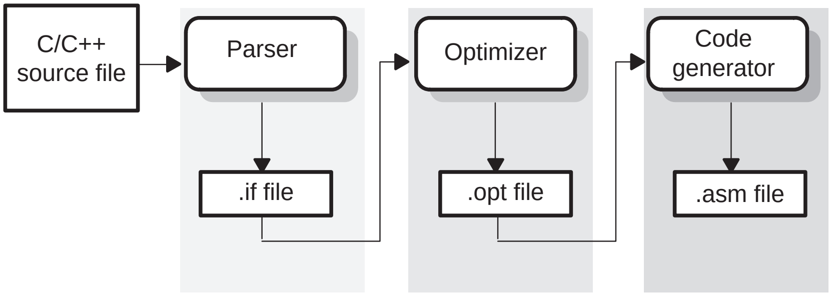
Topic **Page**

3.1 Invoking Optimization

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. High-level optimizations must be used to achieve optimal code.

Figure 3−1 illustrates the execution flow of the compiler with the optimizer and code generator.

Figure 3−1. Compiling a C/C++ Program With Optimization



The easiest way to invoke optimization is to use the cl430 compiler program, specifying the −−opt_level option=*n* (or −O*n)* option on the cl430 command line. The *n* denotes the level of optimization (0, 1, 2, and 3), which controls the type and degree of optimization.

❑ **−−opt_level=0** or **−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

❑ **−−opt_level=1** or **−O1**

Performs all −−opt_level=0 (−O0) optimizations, plus:

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

- **−−opt_level=2** or **−O2**

  Performs all −−opt_level=1 (−O1) optimizations, plus:

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

  The optimizer uses −−opt_level=2 as the default if you use −−opt_level without an optimization level.

- **−−opt_level=3** or **−O3**

  Performs all −−opt_level=2 (−O2) optimizations, plus:

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

  If you use −−opt_level=3 (−O3), see section 3.2, *Performing File-Level Optimization (−−opt_level=3 Option)*, on page 3-4 and section 3.3, *Performing Program-Level Optimization (−−program_level_compile and −−opt_level=3 Options)*, on page 3-6 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 although they are much more effective when the optimizer is used.

3.2 Performing File-Level Optimization (−−opt_level=3 Option)

The −−opt_level=3 option (−O3) instructs the compiler to perform file-level optimization. You can use the −−opt_level=3 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 −−opt_level=3 to perform the indicated optimization:

Table 3−1. Options That You Can Use With −−opt_level=3

If you ...	**Use this option**	**Page**
Have files that redeclare standard library functions	−−std_lib_defined −−std_lib_redefined	3-4
Want to create an optimization information file	−−gen_opt_info=*n*	3-4
Want to compile multiple source files	−−program_level_compile	3-6

3.2.1 Controlling File-Level Optimizations (−−std_lib_func_ Options)

When you invoke the optimizer with the −−opt_level=3 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. Use Table 3−2 to select the appropriate file−level optimization option.

Table 3−2. Selecting a File−Level Optimization Option

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

3.2.2 Creating an Optimization Information File (−−gen_opt_info Option)

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

Table 3−3. Selecting a Level for the −−gen_opt_info Option

If you …	**Use this option**
Do not want to produce an information file, but you used the −−gen_opt_info=1 or −−gen_opt_info=2 option in a command file or an environment variable. The −−gen_opt_info=0 option restores the default behavior of the optimizer.	−−gen_opt_info=0
Want to produce an optimization information file	−−gen_opt_info=1
Want to produce a verbose optimization information file	−−gen_opt_info=2

3.3 Performing Program-Level Optimization (−−program_level_compile and −−opt_level=3 Options)

You can specify program-level optimization by using the −−program_level_compile option with the −−opt_level=3 option (−O3). 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 −−gen_opt_info=2 option to generate an information file. See section 3.2.2, *Creating an Optimization Information File (−−gen_opt_info Option)*, on page 3-5 for more information.

In Code Composer Studio, when the −−program_level_compile option is used, C and C++ files that have the same options are compiled together. However, if any file has a file-specific option that is not selected as a project-wide option, that file is compiled separately. For example, if every C and C++ file in your project has a different set of file-specific options, each is compiled separately, even though program-level optimization has been specified. To compile all C and C++ files together, make sure the files do not have file-specific options. Be aware that compiling C and C++ files together may not be safe if previously you used a file-specific option such as −−aliased_variable.

Note: Compiling Files With the −−program_level_compile and −−keep_asm Options

If you compile all files with the −−program_level_compile and −−keep_asm options, the compiler produces only one .asm file, not one for each corresponding source file.

3.3.1 Controlling Program-Level Optimization (−−call_assumptions Option)

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

Table 3−4. Selecting a Level for the −−call_assumptions Option

If your module …	**Use this option**
Has functions that are called from other modules and global variables that are modified in other modules	−−call_assumptions=0
Does not have functions that are called by other modules but has global variables that are modified in other modules	−−call_assumptions=1
Does not have functions that are called by other modules or global variables that are modified in other modules	−−call_assumptions=2
Has functions that are called from other modules but does not have global variables that are modified in other modules	−−call_assumptions=3

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

Table 3−5. Special Considerations When Using the −−call_assumptions Option

If your option is...	Under these conditions...	Then the −−call_assumptions= level...
Not specified	The −−opt_level=3 optimization level was specified	Defaults to −−call_assumptions=2
Not specified	The compiler sees calls to outside functions under the −−opt_level=3 optimization level	Reverts to −−call_assumptions=0
Not specified	Main is not defined	Reverts to −−call_assumptions=0
−−call_assumptions=1 or −−call_assumptions=2	No function has main defined as an entry point and functions are not identified by the FUNC_EXT_CALLED pragma	Reverts to −−call_assumptions=0
−−call_assumptions=1 or −−call_assumptions=2	No interrupt function is defined	Reverts to −−call_assumptions=0
−−call_assumptions=1 or −−call_assumptions=2	Functions are identified by the FUNC_EXT_CALLED pragma	Remains −−call_assumptions=1 or −−call_assumptions=2
−−call_assumptions=3	Any condition	Remains −−call_assumptions=3

In some situations when you use −−program_level_compile and −−opt_level=3, you *must* use an −−call_assumptions option or the FUNC_EXT_CALLED pragma. See section 3.3.2, *Optimization Considerations When Mixing C/C++ and Assembly*, on page 3-8 for information about these situations.

3.3.2 Optimization Considerations When Mixing C/C++ and Assembly

If you have any assembly functions in your program, you need to exercise caution when using the −−program_level_compile 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 −−program_level_compile option optimizes out those C/C++ functions. To keep these functions, place the FUNC_EXT_CALLED pragma (see section 5.8.6, *The FUNC_EXT_CALLED Pragma*, on page 5-17) 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 −−call_assumptions=*n* option with the −−program_level_compile and −−opt_level=3 options (see section 3.3.1, *Controlling Program-Level Optimization*, on page 3-7).

In general, you achieve the best results through judicious use of the FUNC_EXT_CALLED pragma in combination with −−program_level_compile −−opt_level=3 and −−call_assumptions=1 or −−call_assumptions=2.

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 −−program_level_compile −−opt_level=3 −−call_assumptions=2 to tell the compiler that outside functions do not call C/C++ functions or modify C/C++ variables. See section 3.3.1 for information about the −−call_assumptions=2 option.

If you compile with the −−program_level_compile −−opt_level=3 options only, the compiler reverts from the default optimization level (2) to −−call_assumptions=0. The compiler uses −−call_assumptions=0, 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 −−program_level_compile −−opt_level=3 −−call_assumptions=1.
- Add the volatile keyword to those variables that may be modified by the assembly functions and compile with −−program_level_compile −−opt_level=3 −−call_assumptions=2.

See section 3.3.1 on page 3-7 for information about the −−call_assumptions option.

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

Solution Add the volatile keyword to the C/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 −−program_level_compile −−opt_level=3 −−call_assumptions=2. *Be sure that you use the pragma with all of the entry-point functions.* If you do not, the compiler might remove the entry-point functions that are not preceded by the FUNC_EXT_CALL pragma.
- Compile with −−program_level_compile −−opt_level=3 −−call_assumptions=3. Because you do not use the FUNC_EXT_CALL pragma, you must use the −−call_assumptions=3 option, which is less aggressive than the −−call_assumptions=2 option, and your optimization may not be as effective.

Keep in mind that if you use −−program_level_compile −−opt_level=3 without additional options, the compiler removes the C functions that the assembly functions call. Use the FUNC_EXT_CALLED pragma to keep these functions.

3.4 Using 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.

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

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 to the same object, then the optimizer assumes that the pointers do point to the same object.

The compiler assumes that if the address of a local variable is passed to a function, the function changes the local variable by writing through the pointer. This makes the local variable's address unavailable for use elsewhere after returning. For example, the called function cannot assign the local variable's address to a global variable or return the local variable's address. In cases where this assumption is invalid, use the −−aliased_variable compiler option to force the compiler to assume worst-case aliasing. In worst-case aliasing, any indirect reference can refer to such a variable.

3.6 Automatic Inline Expansion (−−auto_inline Option)

When optimizing with the −−opt_level=3 option (−O3), the compiler automatically inlines small functions. The −−auto_inline=*size* option specifies the size threshold. Any function larger than the *size* threshold is not automatically inlined. You can use the −−auto_inline=*size* option in the following ways:

- If you set the *size* parameter to 0 (−−auto_inline=0), 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 −−gen_opt_info=1 or −−gen_opt_info=2 option) reports the size of each function in the same units that the −−auto_inline option uses.

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

Note: Optimization Level 3 and Inlining

In order to turn on automatic inlining, you must use the −−opt_level=3 option (−O3). The −−opt_level=3 option turns on other optimizations. If you desire the −−opt_level=3 optimizations, but not automatic inlining, use −−auto_inline=0 with the −−opt_level=3 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 −−auto_inline=0 and −−no_inlining options. These options cause the compiler to inline intrinsics only.

3.7 Using Interlist With Optimization

You control the output of the interlist feature when compiling with optimization (the −−opt_level=*n* or −O*n* option) with the −−optimizer_interlist and −−c_src_interlist options.

- The −−optimizer_interlist option interlists compiler comments with assembly source statements.
- The −−c_src_interlist and −−optimizer_interlist options together interlist the compiler comments and the original C/C++ source with the assembly code.

When you use the −−optimizer_interlist option with optimization, the interlist feature 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 −−c_src_interlist option also.

The interlist feature can affect optimized code because it might prevent some optimization from crossing C/C++ statement boundaries. Optimization makes normal source interlisting impractical, because the compiler extensively rearranges your program. Therefore, when you use the −−optimizer_interlist 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 2−1 on page 2-40 compiled with optimization (−−opt_level=2) and the −−optimizer_interlist option. The assembly file contains compiler comments interlisted with assembly code.

Example 3−1. Listing for Compilation With the −−opt_level=2 and −−optimizer_interlist Options

```
main:
;* ----------------------------------------------------------------------------*
        SUB.W     #2,SP
;** 5   ------------------------   printf((const unsigned char *)"Hello, world\n");
        MOV.W     #$C$SL1+0,0(SP)       ; |5|
        CALL      #printf               ; |5|
                                          ; |5|
;** 6   ------------------------   return 0;
        MOV.W     #0,r12                ; |6|
        ADD.W     #2,SP
        RET
```

When you use the −−c_src_interlist and −−optimizer_interlist options with optimization, the compiler inserts its comments and the interlist feature runs before the assembler, merging the original C/C++ source into the assembly file.

Example 3−2 shows the function from Example 2−1 compiled with the optimizer (−−opt_level=2) and the −ss and −−optimizer_interlist options. The assembly file contains optimizer comments and C source interlisted with assembly code.

Example 3−2. Listing for Compilation With the −−opt_level=2, −−optimizer_interlist, and −−c_src_interlist Options

```
main:
;* ---------------------------------------------------------------------------*
        SUB.W     #2,SP
;** 5  -----------------------    printf((const unsigned char *)"Hello, world\n");
;------------------------------------------------------------------------
;   5 | printf ("Hello, world\n");
;------------------------------------------------------------------------
        MOV.W     #$C$SL1+0,0(SP)       ; |5|
        CALL      #printf               ; |5|
                                          ; |5|
;** 6  -----------------------    return 0;
;------------------------------------------------------------------------
;   6 | return 0;
;------------------------------------------------------------------------
        MOV.W     #0,r12                ; |6|
        ADD.W     #2,SP
        RET
```

Note: Impact on Performance and Code Size

The −−c_src_interlist option can have a negative effect on performance and code size.

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. Profiling code that has been built with the −−symdebug:dwarf (or −g) option is not recommended as well, because these options can significantly degrade performance. To remedy these problems, you can use the options described in the following sections to optimize your code in such a way that you can still debug or profile the code.

To debug optimized code, use the −−opt_level option (−O) in conjunction with the symbolic debugging option (−−symdebug:dwarf). The symbolic debugging option generates directives that are used by the C/C++ source-level debugger, but it disables many compiler optimizations. When you use the −−opt_level option (which invokes optimization) with the −−symdebug:dwarf option, you turn on the maximum amount of optimization that is compatible with debugging.

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

Note: Symbolic Debugging Option Affects Performance and Code Size

Using the −−symdebug:dwarf can cause a significant performance and code size degradation of your code. Use this option for debugging only. Using −−symdebug:dwarf when profiling is not recommended.

3.9 What Kind of Optimization Is Being Performed?

The MSP430 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 −−opt_level (−O) compiler options (see section 3.1, *Invoking Optimization*, on page 3-2). However, the code generator performs some optimizations that you cannot selectively enable or disable.

Following is a small sample of 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 run-time-support library functions	3-21
Induction variable optimizations and strength reduction	3-21
Loop-invariant code motion	3-21
Loop rotation	3-21
Tail merging	3-22
Integer division with constant divisor	3-24
never_executed( ) intrinsic	3-24

3.9.1 Cost-Based Register Allocation

The compiler, when enabled with optimization, 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. Those variables whose uses do not 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 branches.

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/C++ compiler output

```
fsm:
;* ----------------------------------------------------------------------------*
        MOV.W     r15,r14                ; |6|
;* ----------------------------------------------------------------------------*
;*   BEGIN LOOP $C$L1
;*
;* ----------------------------------------------------------------------------*
$C$L1:
        ADD.W     #2,r15                 ; |9|
        ADD.W     #2,r14                 ; |9|
        TST.W     0(r14)                 ; |9|
        JNE       $C$L2                  ; |9|
                                           ; |9|
;* ----------------------------------------------------------------------------*
        ADD.W     #2,r15                 ; |10|
        MOV.W     r15,r14                ; |10|
        TST.W     0(r14)                 ; |10|
        JNE       $C$L1                  ; |10|
                                           ; |10|
;* ----------------------------------------------------------------------------*
;*   BEGIN LOOP $C$L2
;* ----------------------------------------------------------------------------*
$C$L2:
        ADD.W     #2,r15                 ; |11|
        TST.W     0(r15)                 ; |11|
        JEQ       $C$L2                  ; |11|
                                           ; |11|
;* ----------------------------------------------------------------------------*
        RET
```

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 (see Example 3−4).

- **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. For example, the expression (a + b) − (c + d) takes six instructions to evaluate; it can be optimized to ((a + b) − c) − d, which takes only four instructions. Operations between constants are folded into single constants. For example, a = (b + 4) − (c + 1) becomes a = b − c + 3 (see Example 3−4).

In Example 3−4, the constant 3, assigned to a, is copy propagated to all uses of a; a becomes a dead variable and is eliminated. The sum of multiplying j by 3 plus multiplying j by 2 is simplified into b = j * 5, which is recognized as a common subexpression. The assignments to c and d are dead and are replaced with their expressions.

Example 3−4. Data Flow Optimizations and Expression Simplification

(a) C source

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

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

(b) C/C++ compiler output

```
simp:
;* --------------------------------------------------------------------------*
        MOV.W     r12,r15               ; |2|
;** 9  -----------------------    C$1 = j*4;
        MOV.B     r15,r14               ; |9|
        RLA.W     r14                   ; |9|
        RLA.W     r14                   ; |9|
;** 9  -----------------------    call(2, C$1, C$1, (j>>3)+(j<<C$1));
        MOV.B     r15,r12               ; |9|
        MOV.W     r14,r13               ; |9|
        CALL      #I_LSL                ; |9|
                                          ; |9|
        MOV.B     r15,r15               ; |9|
        RRA.W     r15                   ; |9|
        RRA.W     r15                   ; |9|
        RRA.W     r15                   ; |9|
        ADD.W     r12,r15
        MOV.W     #2,r12                ; |9|
        MOV.W     r14,r13               ; |9|
        CALL      #call                 ; |9|
                                          ; |9|
;** 9  -----------------------    return;
        RET
```

3.9.6 Inline Expansion of Run-Time-Support Library Functions

If compiling for speed (using the −−opt_for_speed option) the compiler replaces calls to small run-time-support functions, such as to upper( ), with inline code. This saves the overhead associated with a function call as well as provides increased opportunities to apply other optimizations.

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.

3.9.10 Tail Merging

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

In Example 3−5, the addition to a at the end of all three cases is merged into one block. Also, the multiplication by 3 in the first two cases is merged into another block. This results in a reduction of three instructions. In some cases, this optimization adversely affects execution speed by introducing extra branches.

Example 3−5. Tail Merging

(a) C code

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

   return a;
}
```

Example 3−5. Tail Merging (Continued)

(b) C/C++ compiler output

```
main:
;* ---------------------------------------------------------------------------*
        PUSH.W    r10
        MOV.W     r12,r10                ; |2|
;** 3  -----------------------    if ( a < 0 ) goto g5;
        TST.W     r10                    ; |3|
        JL        $C$L2                  ; |3|
                                           ; |3|
;* ---------------------------------------------------------------------------*
;** 8  -----------------------    if ( a ) goto g4;
        TST.W     r10                    ; |8|
        JNE       $C$L1                  ; |8|
                                           ; |8|
;* ---------------------------------------------------------------------------*
;** 10 -----------------------    a = g(a);
        CALL      #g                     ; |10|
                                           ; |10|
        MOV.W     r12,r10                ; |10|
        JMP       $C$L3                  ; |12|
                                           ; |12|
;* ---------------------------------------------------------------------------*
$C$L1:
;**-----------------------g4:
;** 14 -----------------------    a += f(a);
        CALL      #f                     ; |14|
                                           ; |14|
        JMP       $C$L4                  ; |14|
                                           ; |14|
;* ---------------------------------------------------------------------------*
$C$L2:
;**-----------------------g5:
;** 5  -----------------------    a = -a;
        INV.W     r10                    ; |5|
        ADD.W     #1,r10                 ; |5|
;* ---------------------------------------------------------------------------*
$C$L3:
;** 6  -----------------------    a += f(a)*3;
        MOV.W     r10,r12                ; |6|
        CALL      #f                     ; |6|
                                           ; |6|
        MOV.W     r12,r15                ; |6|
        RLA.W     r12                    ; |6|
        ADD.W     r15,r12
;* ---------------------------------------------------------------------------*
$C$L4:
        ADD.W     r12,r10
;**-----------------------g6:
;** 16 -----------------------    return a;
        MOV.W     r10,r12                ; |16|
        POP       r10
```

3.9.11 Integer Division With Constant Divisor

The optimizer attempts to rewrite integer divide operations with constant divisors. The integer divides are rewritten as a multiply with the reciprocal of the divisor. This occurs at level −−opt_level=2 and higher. You must also compile with the −−opt_for_speed option, which selects compile for speed over space.

3.9.12 Never_Executed( ) Intrinsic

The _never_executed( )intrinsic can be used to assert to the compiler that a switch expression can only take on values represented by the case labels within a switch block. This assertion enables the compiler to avoid generating test code for handling values not specified by the switch case labels. This assertion is specifically suited for handling values that characterize a vector generator. See section 6.8.2 on page 6-27 for details on the _never_executed( ) intrinsic.

Linking C/C++ Code

The MSP430 C/C++ compiler and assembly language tools provide two methods for linking your programs:

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

This chapter describes how to invoke the link step with each method. It also discusses special requirements of linking C/C++ code, including the run-time-support libraries, specifying the initialization model, and allocating the program into memory. For a complete description of the link step, see the *MSP430 Assembly Language Tools User’s Guide*.

Topic **Page**

4.1 Invoking the Link Step (−−run_linker Option)

This section explains how to invoke the link step after you have compiled and assembled your programs: as a separate step or as part of the compile step.

4.1.1 Invoking the Link Step As a Separate Step

This is the general syntax for linking C/C++ programs as a separate step:

cl430 −−run_linker {**−−ram_model**|**−−rom_model**} *filenames* [*options*] [**−−output_file=***name*.out] [*lnk.cmd*] **−library=***library*

cl430 −−run_linker	The command that invokes the link step
−−ram_model| **−−rom_model**	Options that tell the link step to use special conventions defined by the C/C++ environment. When you use cl430 −−run_linker, you must use −−rom_model or −−ram_model. The −−rom_model option uses automatic variable initialization at run time; the −−ram_model option uses automatic variable initialization at load time.
options	Options affect how the link step handles your object files. Options can appear anywhere on the command line or in a link step command file. (Options are discussed in section 4.2, *Link Options*.)
filenames	Names of object files, link step command files, or archive libraries. The default extension for all input files is *.obj*; any other extension must be explicitly specified. The link step can determine whether the input file is an object or ASCII file that contains link step commands. The default output filename is *a.out*, unless you use the −−output_file option to name the output file.
−−output_file	The −−output_file= *name.out* option names the output file.
lnk.cmd	Contains options, filenames, directives, or commands for the link step.
−−library	The −−library=*library* option identifies the appropriate archive library containing C/C++ run-time-support and floating-point math functions. (The −−library option tells the link step that a file is an archive library.) You can use the libraries included with the compiler, or you can create your own run-time-support library. If you have specified a run-time-support library in a link step command file, you do not need this parameter.

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

The link step uses a default allocation algorithm to allocate your program into memory. You can use the MEMORY and SECTIONS directives in the link step command file to customize the allocation process. For more information, see the *MSP430 Assembly Language Tools User's Guide*.

You can link a C/C++ program consisting of modules prog1.obj, prog2.obj, and prog3.obj with an executable filename of prog.out with the command:

```
cl430 --run_linker --rom_model prog1 prog2 prog3 --output=prog.out
      --library=rts430.lib
```

4.1.2 Invoking the Link Step as Part of the Compile Step

This is the general syntax for linking C/C++ programs as part of the compile step:

cl430 [*options*] *filenames* **−−run_linker** {**−−ram_model**|**−−rom_model**} *filenames* [*options*] [**−−output_file=***name*.out] [*lnk.cmd*] **−library=***library*

The −−run_linker option divides the command line into the compiler options (the options before −−run_linker) and the link options (the options following −−run_linker). The −−run_linker option must follow all source files and compiler options on the command line.

All arguments that follow −z on the command line are passed to the link step. These arguments can be link step command files, additional object files, link options, or libraries. These arguments are the same as described in section 4.1.1, *Invoking the Link Step As a Separate Step*, on page 4-2.

All arguments that precede −z on the command line are compiler arguments compiler. These arguments can be C/C++ source files, assembly files, or compiler options. For a description of these arguments, see section 2.2, *Invoking the C Compiler*, on page 2-4.

You can compile and link a C/C++ program consisting of modules prog1.c, prog2.c, and prog3.c with an executable filename of prog.out with the command:

```
cl430 prog1.c prog2.c prog3.c --run_linker --rom_model --output_file=prog.out
      --library=rts430.lib
```

Note: Order of Processing Arguments in the Link Step

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

1) Object filenames from the command line
2) Arguments following the −−run_linker option on the command line
3) Arguments following the −−run_linker option from the MSP430_C_OPTION environment variable

4.1.3 Disabling the Link Step (−compile_only Option)

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

4.2 Link Step Options

All command-line input following the −−run_linker option is passed to the link step as parameters and options. Table 4−1 lists the options that control the link step, along with their aliases and descriptions of their effects.

Table 4−1. Link Step Options Summary

Option	Alias	Description
−−absolute_exe	−a	Produces an absolute, executable module. This is the default; if neither −−absolute_exe nor −−relocatable is specified, the link step acts as if −−absolute_exe is specified.
−ar		Produces a relocatable, executable object module
−−arg_size=*size*	−−args	Allocates memory to be used by the loader to pass arguments from the command line of the loader to the program. The link step allocates *size* bytes in an uninitialized .args section. The __c_args__ symbol contains the address of the .args section.
−−disable_clink	−j	Disables conditional linking. The C/C++ compiler automatically generates the .clink assembler directive with every section that is a candidate for conditional linking; the −−disable_link option overrides this default. For more information about conditional linking and the .clink assembler directive, see the *MSP430 Assembly Language Tools User's Guide*.
−−entry_point=*global_symbol*	−e	Defines a *global_symbol* that specifies the primary entry point for the output module.
−−fill_value=*value*	−f	Sets the default fill value for holes within output sections; *value* must be a 16-bit constant.
−−heap_size=*size*	−heap −−heap	Sets heap size (for the dynamic memory allocation) to *size* bytes and defines a global symbol that specifies the heap size. The default is 128 bytes.
−−ignore_align_flags	−k	Ignores alignment flags in input sections
−−library=*filename*	−l	Names an archive library file as link step input and instructs the link step to use the C_DIR environment variable and paths specified with the −−search_path option to help find the archive library; *filename* is an archive library name and must follow operating system conventions.
−−linker_help	−help −−help	Prints out a full listing of options and their meanings. When −−linker_help is followed by another option, detailed information on that option is displayed.

Table 4−1. Link Step Options Summary (Continued)

Option	Alias	Description
−−make_global=*symbol*	−g	Defines a *global_symbol* as global even if the global symbol has been made static with the −−make_static link option.
−−make_static	−h	Makes all global symbols static.
−−map_file=*filename*	−m	Produces a map or listing of the input and output sections, including holes, and places the listing in *filename.* The filename you specify must follow operating system conventions.
−−no_sym_merge	−b	Disables merge of symbolic debugging information. The link step keeps the duplicate entries of symbolic debugging information commonly generated when a C program is compiled for debugging.
−−no_sym_table	−s	Strips symbol table information and line number entries from the output module.
−−output_file=*filename*	−o	Names the executable output module. The *filename* must follow operating system conventions. If the −−output_file option is not used, the default filename is a.out.
−−priority	−priority	Satisfies each unresolved reference by the first library that contains a definition for that symbol
−−ram_model	−cr	Autoinitializes variables at load time
−−relocatable	−r	Retains relocation entries in the output module
−−reread_libs	−x	Forces rereading of libraries to resolve backward symbol references
−−rom_model	−c	Autoinitializes variables at run time
−−run_abs	−abs	Produces an absolute listing file. You must use the −−opt_level option (after −−run_linker) to specify the .out file for the absolute lister, even if you use a link step command file that already uses −−opt_level.
−−search_path=*directory*	−I	Alters the library-search algorithm to look in *directory* before looking in the default location(s). The −−search_path option must appear before the −−library option. The directory you specify must follow operating system conventions.
−−stack_size=*size*	−stack	Sets the C/C++ system stack size to *size* bytes and defines a global symbol that specifies the stack size. The default is 2048 bytes.
−−undef_sym=*symbol*	−U	Places the unresolved external symbol *symbol* into the output module's symbol table

Table 4−1. Link Step Options Summary (Continued)

Option	Alias	Description
−−use_hw_mpy		Replaces all references to the default multiply routine with the version of the multiply routine that uses the hardware multiplier support
−−warn_sections	−w	Displays a message when an undefined output section is created
−−xml_link_info		Generates an XML link information file. This option causes the link step to generate a well-formed XML file containing detailed information about the result of a link. The information included in this file includes all of the information that is currently produced in a link step generated map file.

For more information on link options, see the *Link Step Description* chapter of the *MSP430 Assembly Language Tools User's Guide.*

4.3 Controlling the Linking Process

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

- Include the compiler's run-time-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 link step command file.

For more information about how to operate the link step, see the *Link Step Description* chapter of the *MSP430 Assembly Language Tools User's Guide.*

4.3.1 Linking With Run-Time-Support Libraries

You must link all C/C++ programs with a run-time-support library. The library contains standard C/C++ functions as well as functions used by the compiler to manage the C/C++ environment. You must use the −−library link option to specify the run-time-support library (rts430.lib or rts430x.lib) to use. The −−library option also tells the link step to look at the −−search_path options and then the MSP430_C_DIR environment variable to find an archive path or object file.

To use the −−library option, type on the command line:

cl430 −−run_linker {**−−ram_model**|**−−rom_model**} *filenames* **−library=***library*

Generally, the libraries should be specified as the last filenames on the command line because the link step 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 −−reread_libs option to force the link step to reread all libraries until references are resolved. Wherever you specify a library as link step input, the link step includes and links only those library members that resolve undefined references.

4.3.2 Run-Time Initialization

You must link all C/C++ programs with code to initialize and execute the program called a *bootstrap routine*. The bootstrap routine is responsible for the following tasks:

1) Set up the stack
2) Process the .cinit run-time initialization table to autoinitialize global variables (when using the −−rom_model option)
3) Calls all global constructors (.pinit) for C++
4) Calls main
5) Calls exit when main returns

A sample bootstrap routine is _c_int00, provided in boot.obj in the run-time-support libraries. The *entry point* is usually set to the starting address of the bootstrap routine.

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

Note: The _c_int00 Symbol

If you use the −−rom_model or −−ram_model link option, _c_int00 is automatically defined as the entry point for the program.

4.3.3 Initialization By the Interrupt Vector

If your program begins running from load time, you must set up the reset vector to branch to _c_int00. This causes boot.obj to be loaded from the library and your program is initialized correctly. The boot.obj places the address of _c_int00 into a section named .reset. This section can then be allocated at the reset vector location using the link step.

4.3.4 Global Object Constructors

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 other global variables and before main( ) is called. Global destructors are invoked during exit( ) similarly to functions registered through atexit( ).

Section 6.9.5, *Initialization Tables*, discusses the format of the .pinit table.

4.3.5 Specifying the Type of Global Variable Initialization

The C/C++ compiler produces data tables for autoinitializing global variables. Section 6.9.5, *Initialization Tables*, on page 6-32 discusses the format of these tables. These tables are in a named section called *.cinit*. The initialization tables are used in one of the following ways:

- ❑ Autoinitializing variables at run time. Global variables are initialized at *run time*. Use the −−rom_model link option (see section 6.9.6, *Autoinitialization of Variables at Run Time*, on page 6-34).
- ❑ Initializing variables at load time. Global variables are initialized at *load time*. Use the −−ram_model link option (see section 6.9.7, I*nitialization of Variables at Load Time*, on page 6-35).

When you link a C/C++ program, you must use either the −−rom_model or the −−ram_model link option. These options tell the link step to select autoinitialization at run time or load time. When you compile and link programs, the −−rom_model link option is the default; if used, the −−rom_model link option must follow the −−run_linker option (see section 4.1, *Invoking the Link Step (−−run_linker Option)* on page 4-2). The following list outlines the linking conventions used with −−rom_model or −−ram_model:

- ❑ The symbol _c_int00 is defined as the program entry point; it identifies the beginning of the C boot routine in boot.obj. When you use −−rom_model or −−ram_model, _c_int00 is automatically referenced; this ensures that boot.obj is automatically linked in from the run-time-support library.
- ❑ The *.cinit* output section is padded with a termination record so that the loader (load-time initialization) or the boot routine (run-time initialization) knows when to stop reading the initialization tables.
- ❑ The *.pinit* output section is padded with a termination record.
- ❑ When initializing at load time (the −−ram_model link option), the following occur:
  - ■ The link step sets the symbol cinit to −1. This indicates that the initialization tables are not in memory, so no initialization is performed at run time.

  - The STYP_COPY flag 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 link step does not allocate space in memory for the .cinit section.

- When autoinitializing at run time (−−rom_model link option), the link step defines the symbol cinit as the starting address of the .cinit section. The boot routine uses this symbol as the starting point for autoinitialization.

- The link step defines the symbol pinit as the starting address of the .pinit section. The boot routine uses this symbol as the beginning of the table of global constructors.

Note: Boot Loader

A loader is not included as part of the MSP430 C/C++ compiler tools.

4.3.6 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 4−2 summarizes the sections.

Table 4−2. Sections Created by the Compiler

(a) Initialized sections

Name	Contents
.cinit	Tables for explicitly initialized global and static variables
.const	Global and static const variables that are explicitly initialized and string literals
.pinit	Tables for global object constructors
.text	Executable code

(b) Uninitialized sections

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

When you link your program, you must specify where to allocate the sections in memory. In general, initialized sections are linked into ROM or RAM; uninitialized sections are linked into RAM. See section 6.1.1, *Sections*, on page 6-2 for a complete description of how the compiler uses these sections. The link step provides MEMORY and SECTIONS directives for allocating sections. For more information about allocating sections into memory, see the link step description chapter of the *MSP430 Assembly Language Tools User's Guide*.

4.3.7 A Sample Link Step Command File

Example 4−1 shows a typical link step command file that links a 32-bit C program. The command file in this example is named lnk32.cmd and lists several link options:

−−rom_model	Tells the link step to use autoinitialization at run time
−−stack_size	Tells the link step to set the C stack size at 0x140 bytes
−−heap_size	Tells the link step to set the heap size to 0x120 bytes
−−library	Tells the link step to use an archive library file, rts430.lib

To link the program, enter:

cl430 −−run_linker *object_file(s)* **−−output_file=***file* **−−map_file=***file* lnk.cmd

Example 4−1. Link Step Command File

```
--rom_model
--stack_size=0x0140
--heap_size=0x120
--library=rts430.lib

/****************************************************************************/
/* SPECIFY THE SYSTEM MEMORY MAP                                            */
/****************************************************************************/

MEMORY
{
    SFR(R)            : origin = 0x0000, length = 0x0010
    PERIPHERALS_8BIT : origin = 0x0010, length = 0x00F0
    PERIPHERALS_16BIT: origin = 0x0100, length = 0x0100
    RAM(RW)           : origin = 0x0200, length = 0x0800
    INFOA             : origin = 0x1080, length = 0x0080
    INFOB             : origin = 0x1000, length = 0x0080
    FLASH             : origin = 0x1100, length = 0xEEE0
    VECTORS(R)        : origin = 0xFFE0, length = 0x001E
    RESET             : origin = 0xFFFE, length = 0x0002
}
/***************************************************************************/
/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY                             /
/***************************************************************************/

SECTIONS
{
    .bss     : {} > RAM                /* GLOBAL & STATIC VARS              */
    .sysmem  : {} > RAM                /* DYNAMIC MEMORY ALLOCATION AREA    */
    .stack   : {} > RAM                /* SOFTWARE SYSTEM STACK             */

    .text    : {} > FLASH              /* CODE                              */
    .cinit   : {} > FLASH              /* INITIALIZATION TABLES             */
    .const   : {} > FLASH              /* CONSTANT DATA                     */
    .cio     : {} > RAM                /* C I/O BUFFER                      */

    .pinit   : {} > RAM                /* C++ CONSTRUCTOR TABLES            */

    .intvecs : {} > VECTORS            /* MSP430 INTERRUPT VECTORS          */
    .reset   : {} > RESET              /* MSP430 RESET VECTOR               */
```

4.3.8 Using Function Subsections (−−gen_func_subsections Compiler Option)

When the link step places code into an executable file, it allocates all the functions in a single source file as a group. This means that if any function in a file needs to be linked into an executable, then all the functions in the file are linked in. This can be undesirable if a file contains many functions and only a few are required for an executable.

This situation may exist in libraries where a single file contains multiple functions, but the application only needs a subset of those functions. An example is a library .obj file that contains a signed divide routine and an unsigned divide routine. If the application requires only signed division, then only the signed divide routine is required for linking. By default, both the signed and unsigned routines are linked in since they exist in the same .obj file.

The −−gen_func_subsections compiler option remedies this problem by placing each function in a file in its own subsection. Thus, only the functions that are referenced in the application are linked into the final executable. This can result in an overall code size reduction.

Chapter 5

MSP430 C and C++ Languages

The MSP430 C/C++ compiler supports the C language standard developed by a committee of of the International Organization for Standardization (ISO) to standardize the C programming language.

The C++ language supported by the MSP430 is defined in the ISO/IEC 14882−1998 C++ Standard and described in the *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 MSP430 C

ISO 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 ISO C standard is described in the International Standard ISO/IEC 9899 (1989)—Programming languages—C (The C Standard).

The ANSI/ISO standard identifies certain features of the C language that are affected by characteristics of the target processor, run-time 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 MSP430 C/C++ compiler.

The following list identifies all such cases and describes the behavior of the MSP430 C compiler in each case. Each description also includes a reference to more information. Many of the references are to the formal ISO C standard or to K&R.

5.1.1 Identifiers and Constants

- All 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.
(ISO 6.1.2, K&R A2.3)

- The source (host) and execution (target) character sets are assumed to be ASCII. There are no multibyte characters.
(ISO 5.2.1, K&R A12.1)

- Hex or octal escape sequences in character or string constants may have values up to 32 bits. (ISO 6.1.3.4, K&R A2.5.2)

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

  `'abc' == 'c'` (ISO 6.1.3.4, K&R A2.5.2)

- A long long integer constant can have an ll or LL suffix. If the suffix is not used, the value of the constant will determine the type of the constant.

5.1.2 Data Types

- For information about the representation of data types, see section 5.3. (ISO 6.1.2.5, K&R A4.2)
- The type size_t, which is the result of the *sizeof* operator, is unsigned int. (ISO 6.3.3.4, K&R A7.4.8)
- The type ptrdiff_t, which is the result of pointer subtraction, is int. (ISO 6.3.6, K&R A7.7)

5.1.3 Conversions

- Float-to-integer conversions truncate toward 0. (ISO 6.2.1.3, 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. (ISO 6.3.4, 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
  ```
  (ISO 6.3.5, 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. (ISO 3.3.7, K&R A7.8)

5.1.5 Declarations

- The *register* storage class is effective for all chars, shorts, ints, and pointer types. For more information, see section 5.6, *Register Variables*, on page 5-11. (ISO 3.5.1, K&R A2.1)
- Structure members are packed into 16-bit words. (ISO 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. (ISO 3.5.2.1, K&R A8.3)

5.1.6 Preprocessor

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

  The following pragmas *are* supported:

  - CODE_SECTION
  - DATA_ALIGN
  - DATA_SECTION
  - FUNC_CANNOT_INLINE
  - FUNC_EXT_CALLED
  - FUNC_NO_GLOBAL_ASG
  - FUNC_NO_IND_ASG
  - FUNC_IS_PURE
  - FUNC_NEVER_RETURNS
  - INTERRUPT

  For more information on pragmas, see section 5.8, *Pragma Directives*, on page 5-13.

- The preprocessor supports the _Pragma operator. For more information, see section 5.9, *The _Pragma Operator*, on page 5-21.

5.2 Characteristics of MSP430 C++

The MSP430 compiler supports C++ as defined in the ISO/IEC 14882−1998 C++ standard. The exceptions to the standard are as follows:

- Complete C++ standard library support is not included. C subset and basic language support is included.
- These C++ headers for C library facilities are not included:
  - <clocale>
  - <csignal>
  - <cwchar>
  - <cwctype>
- These C++ headers are the only C++ standard library header files included:
  - <new>
  - <typeinfo>
  - <ciso646>
- Run-time type information (RTTI) is disabled by default. RTTI can be enabled with the −−rtti compiler option.
- 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 [temp.res] and [temp.dep] of the standard, is not implemented.
- Template parameters are not implemented.
- The export keyword for templates is not implemented.
- A typedef of a function type cannot include member function cv-qualifiers.
- A partial specialization of a class member template cannot be added outside of the class definition.

5.3 Data Types

Table 5−1 lists the size, representation, and range of each scalar data type for the MSP430 compiler. Many of the range values are available as standard macros in the header file limits.h. For more information, see section 7.3.5, *Limits (float.h/cfloat and limits.h/climits)*, on page 7-18.

Table 5−1. MSP430 C/C++ Data Types

			Range	
Type	**Size**	**Representation**	**Minimum Value**	**Maximum Value**
signed char	8 bits	ASCII	−128	127
char, unsigned char, bool	8 bits	ASCII	0	255
short, signed short	16 bits	2s complement	−32 768	32 767
unsigned short, wchar_t	16 bits	Binary	0	65 535
int, signed int	16 bits	2s complement	−32 768	32 767
unsigned int	16 bits	Binary	0	65 535
long, signed long	32 bits	2s complement	−2 147 483 648	2 147 483 647
unsigned long	32 bits	Binary	0	4 294 967 295
enum	16 bits	2s complement	−32768	32 767
float	32 bits	IEEE 32-bit	1.175495e−38	3.40282346e+38
double	32 bits	IEEE 32-bit	1.175495e−38	3.40282346e+38
long double	32 bits	IEEE 32-bit	1.175495e−38	3.40282346e+38
pointers, references, pointer to data members	16 bits	Binary	0	0xFFFF
MSP function pointers	16 bits	Binary	0	0xFFFF
MSPX function pointers†	20 bits	Binary	0	0xFFFFF

† MSPX devices are specified by −−silicon_version=mspx

5.4 Keywords

The MSP430 C/C++ compiler supports the standard const and volatile keywords. In addition, the MSP430 C/C++ compiler extends the C language through the support of the interrupt keyword.

5.4.1 The const Keyword

The MSP430 C/C++ compiler supports the ANSI/ISO 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 its value is 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 has automatic storage (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 interrupt Keyword

The MSP430 C/C++ compiler extends the C language by adding the interrupt keyword to specify that a function is to be treated as an interrupt function.

Functions that handle interrupts 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 only with a function that is defined to return void and that has no parameters. (An exception is software interrupts. For more information, see section 6.6.5, *Using Software Interrupts*.) 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/ISO mode (using the −−strict_ansi compiler option).

5.4.3 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 5−1, the restrict keyword is used to tell the compiler that the function func1 is never called with the the pointers a and b pointing to objects that overlap in memory. The user is promising that accesses through a and b never conflict; this means that a write through one pointer cannot affect a read from any other. The precise semantics of the restrict keyword are described in the 1999 version of the ISO C standard.

Example 5−1. Use of the restrict Type Qualifier With Pointers

```
void func1(int * restrict a, int * restrict b)
{
   int i
   for(i=0;i<64;i++)*a++=*b++;
}
```

Example 5−2 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 5−2. Use of the restrict Type Qualifier With Arrays

```
void func2(int c[restrict], int d[restrict])
{
   int i;

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

5.4.4 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 does not optimize out any references to volatile variables.

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

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

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

```
volatile unsigned int *ctrl
```

5.5 C++ Exception Handling

The compiler supports all the C++ exception handling features as defined by the ANSI/ISO 14882 C++ Standard. More details are discussed in *The C++ Programming Language, Third Edition* by Bjarne Stroustrup.

The compiler −−exceptions option enables exception handling. The compiler's default is no exception handling support.

For exceptions to work correctly, all C++ files in the application must be compiled with the −−exceptions option, regardless of whether exceptions occur in a particular file. Mixing exception-enabled object files and libraries with object files and libraries that do not have exceptions enabled can lead to undefined behavior. Also, when using −−exceptions, you need to link with run-time-support libraries whose name contains _eh. These libraries contain functions that implement exception handling.

Using −−exceptions causes increases in code size and execution time, even if no exceptions are thrown.

5.6 Register Variables

The C/C++ compiler allows the use of the keyword register on global, and local register variables and parameters. This section describes the compiler implementation for this qualifier.

5.6.1 Local 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. This may cause excessive movement of register contents to memory.

Any object with a scalar type (integer, floating point, or pointer) or a structure/union type whose size is less than or equal to 32 bits 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 6-11.

5.7 The asm Statement

The MSP430 C/C++ compiler can embed MSP430 assembly language instructions or directives directly into the assembly language output of the compiler. This capability is standard in the C++ language and is an extension for C. 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 *MSP430 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.

Use the alternate statement __asm(”*assembler text*”); if you are writing code for strict ANSI/ISO C mode (using the −−strict_ansi compiler option).

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.

5.8 Pragma Directives

Pragma directives tell the compiler's preprocessor how to treat functions. The MSP430 C/C++ compiler supports the following pragmas:

- BIS_IE1_INTERRUPT
- CODE_SECTION
- DATA_ALIGN
- DATA_SECTION
- FUNC_CANNOT_INLINE
- FUNC_EXT_CALLED
- FUNC_IS_PURE
- FUNC_NEVER_RETURNS
- FUNC_NO_GLOBAL_ASG
- FUNC_NO_IND_ASG
- INTERRUPT

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

5.8.1 The BIS_IE1_INTERRUPT Pragma

The BIS_IE1_INTERRUPT pragma treats the named function as an interrupt routine. Additionally, the compiler generates a BIS operation on the IE1 special function register upon function exit. The *mask* value, which must be an 8-bit constant literal, is logically OR'ed with the IE1 SFR, just before the RETI instruction. The compiler assumes the IE1 SFR is mapped to address 0x0000.

The syntax of the pragma in C is:

#pragma BIS_IE1_INTERRUPT (*func*, *mask***)** [;]

The syntax of the pragma in C++ is:

#pragma BIS_IE1_INTERRUPT (*mask***)** [;]

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

5.8.2 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 5−3 demonstrates the use of the CODE_SECTION pragma.

Example 5−3. 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"
    .align 2
    .clink
    .global   funcA

;******************************************************************************
;* FUNCTION NAME: funcA                                                       *
;*                                                                            *
;*   Regs Modified     : SP,SR,r12                                            *
;*   Regs Used         : SP,SR,r12                                            *
;*   Local Frame Size  : 0 Args + 4 Auto + 0 Save = 4 byte                    *
;******************************************************************************
funcA:
;* ---------------------------------------------------------------------------*
        SUB.W     #4,SP
        MOV.W     r12,0(SP)              ; |4|
        MOV.W     0(SP),2(SP)            ; |6|
        MOV.W     2(SP),r12              ; |6|
        ADD.W     #4,SP
        RET
```

Example 5−3. Using the CODE_SECTION Pragma (Continued)

(c) C++ source file

```
#pragma CODE_SECTION(”codeB”)
int i_arg(int x) { return 1; }
int f_arg(float x) { return 2; }
```

(d) Assembly source file

```
      .sect ”codeB”
      .align      2
      .clink
      .global    i_arg__Fi

;******************************************************************************
;* FUNCTION NAME: i_arg(int)                                                  *
;*                                                                            *
;*   Regs Modified     : SP,SR,r12                                            *
;*   Regs Used         : SP,SR,r12                                            *
;*   Local Frame Size  : 0 Args + 2 Auto + 0 Save = 2 byte                   *
;******************************************************************************
i_arg__Fi:
;* --------------------------------------------------------------------------*
        SUB.W      #2,SP
        MOV.W      r12,0(SP)              ; |2|
        MOV.W      #1,r12                 ; |2|
        ADD.W      #2,SP
        RET
      .sect ”.text”
      .align      2
      .clink
      .global    f_arg__Ff

;******************************************************************************
;* FUNCTION NAME: f_arg(float)                                                *
;*                                                                            *
;*   Regs Modified     : SP,SR,r12                                            *
;*   Regs Used         : SP,SR,r12,r13                                        *
;*   Local Frame Size  : 0 Args + 4 Auto + 0 Save = 4 byte                   *
;******************************************************************************
f_arg__Ff:
;* --------------------------------------------------------------------------*
        SUB.W      #4,SP
        MOV.W      r12,0(SP)              ; |3|
        MOV.W      r13,2(SP)              ; |3|
        MOV.W      #2,r12                 ; |3|
        ADD.W      #4,SP
        RET
```

5.8.3 The DATA_ALIGN Pragma

The DATA_ALIGN pragma aligns the *symbol* to an alignment boundary. The alignment boundary is the maximum of the symbol's default alignment value or the value of the *constant* in bytes. The constant must be a power of 2.

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***);**

5.8.4 The DATA_SECTION Pragma

The DATA_SECTION pragma allocates space for the applied *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 5−4. 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) Resulting assembly source file

```
          .global  bufferA
          .bss     bufferA,512,2
          .global  bufferB
bufferB:  .usect   "my_sect",512,2
```

5.8.5 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.

5.8.6 The FUNC_EXT_CALLED Pragma

When you use program-level optimization (file-level optimization, using the −−opt_level=3 option (−O3), in which the entire program is in one file), the compiler removes any function that is not called, directly or indirectly, by main(). However, 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 that these C/C++ functions call. These functions act as entry points into C/C++.

The syntax of the pragma in C is:

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

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

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. For more information, see section 3.3.2, *Optimization Considerations When Mixing C/C++ and Assembly*, on page 3-8.

5.8.7 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.

5.8.8 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.

5.8.9 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.

5.8.10 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.

5.8.11 The INTERRUPT Pragma

The INTERRUPT pragma enables you to handle interrupts directly with C/C++ code. This pragma specifies that the function to which it is applied is an interrupt.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma INTERRUPT [;]

In C, the argument *func* is the name of the function that is an interrupt. 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 interrupt does not need to conform to a naming convention.

5.9 The _Pragma Operator

The MSP430 C/C++ compiler supports the C99 preprocessor _Pragma() operator. This preprocessor operator is similar to #pragma directives. However, _Pragma can be used in preprocessing macros (#defines).

The syntax of the operator is:

_Pragma (”*string_literal*”) [;]

The argument *string_literal* is interpreted in the same way the tokens following a #pragma directive are processed. The *string_literal* must be enclosed in quotes. A quotation mark that is part of the string_literal must be preceded by a backward slash.

You can use the _Pragma operator to express #pragma directives in macros. For example, the DATA_SECTION syntax:

#pragma DATA_SECTION(*func*,”*section*”)

Is represented by the _Pragma() operator syntax:

_Pragma (”DATA_SECTION(*func*,\”*section*\”)”)

The following code illustrates using _Pragma to specify the DATA_SECTION pragma in a macro:

```
...
#define EMIT_PRAGMA(x) _Pragma(#x)
#define COLLECT_DATA(var) EMIT_PRAGMA(DATA_SECTION(var,”mysection”))

COLLECT_DATA(x)
int x;

...
```

The EMIT_PRAGMA macro is needed to properly expand the quotes that are required to surround the *section* argument to the DATA_SECTION pragma.

5.10 C++ and Name Demangling

For C++ the compiler transforms the names of externally visible identifiers when creating their linknames. Mangling is the process of embedding a function's signature (the number and types of its parameters) into its name. The mangling algorithm used closely follows that described in *The C++ Annotated Reference Manual* (ARM). Mangling allows function overloading, operator overloading, and type-safe linking.

The general form of a C++ linkname for a 32-bit 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
```

this is the resulting assembly:

```
_ _foo_ _Fi;
```

The linkname of foo is _ _foo_ _Fi, indicating that foo is a function that takes a single argument of type int. To aid inspection and debugging, a name demangling utility is provided (see Chapter 9, *C++ Name Demangler*) that demangles names into those found in the original C++ source.

5.11 Initializing Static and Global Variables

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

Note: Variables Are Not Preinitialized Automatically

Your application must preinitialize all variables. The C/C++ compiler makes no provision for preinitializing variables.

5.11.1 Initializing Static and Global Variables With the Link Step

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

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

Because the link step 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 previously initializes .bss to 0 only at load time, and not at system reset or power-up. To make them 0 at runtime, explicitly initialize the variables in your code.

For more information about link step command files and the SECTIONS directive, see the link step description chapter in the *MSP430 Assembly Language Tools User's Guide*.

5.11.2 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 5.11, *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 link step can be used to place the .const section in ROM.

You can use the DATA_SECTION pragma to put the variable in a section other than .const. For example, the following C code:

```
#pragma DATA_SECTION (var, ".mysect");
   const int zero=0;
```

is compiled into this assembly code:

```
   .sect .mysect
zero
   .word 0
```

5.12 Changing the Language Mode (−−kr_compatible, −−relaxed_ansi, and −−strict_ansi Options)

The −−kr_compatible, −−relaxed_ansi, and −−strict_ansi 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/ISO mode
- K&R C mode
- Relaxed ANSI/ISO mode
- Strict ANSI/ISO mode

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

For C++ code, ANSI/ISO mode designates the latest supported working paper. K&R C mode does not apply to C++ code.

5.12.1 Compatibility With K&R C (−−kr_compatible Option)

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

There are subtle changes, however, in the language that can affect existing code. Appendix C in K&R summarizes the differences between ANSI/ISO 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 MSP430 ANSI/ISO C compiler, the compiler has a K&R C option (−−kr_compatible) that modifies some semantic rules of the language for compatibility with older code. In general, the −−kr_compatible option relaxes requirements that are stricter for ANSI/ISO C than for K&R C. The −−kr_compatible option does not disable any new features of the language such as function prototypes, enumerations, initializations, or preprocessor constructs. Instead, −−kr_compatible simply liberalizes the ANSI/ISO rules without revoking any of the features.

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

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

```
int *p;
char *q = p;    /* error without --kr_compatible, */
                /* warning with --kr_compatible */
```

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

```
a;              /* illegal unless --kr_compatible used */
```

- ANSI/ISO 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 C, 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 --kr_compatible used, OK if not */
```

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

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

```
extern int a;
static int a;   /* illegal unless --kr_compatible used */
```

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

```
char c = '\q';  /* same as 'q' if --kr_compatible used, */
                /* error  if not */
```

- ANSI/ISO specifies that bit fields must be of type int or unsigned. With −−kr_compatible, bit fields can be legally declared with any integral type. For example:

```
struct s
{
   short f : 2; /* illegal unless --kr_compatible used */
};
```

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

```
enum { a, b, c, }; /* illegal unless --kr_compatible used */
```

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

```
#endif NAME     /* illegal unless --kr_compatible used */
```

5.12.2 Enabling Strict ANSI/ISO Mode and Relaxed ANSI/ISO Mode (−−strict_ansi and −−relaxed_ansi Options)

Use the −−strict_ansi option when you want to compile under strict ANSI/ISO mode. In this mode, error messages are provided when non-ANSI/ISO 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 −−relaxed_ansi option when you want the compiler to ignore strict ANSI/ISO violations rather than emit a warning (as occurs in normal ANSI/ISO mode) or an error message (as occurs in strict ANSI/ISO mode). In relaxed ANSI/ISO mode, the compiler accepts extensions to the ANSI/ISO C standard, even when they conflict with ANSI/ISO C.

5.12.3 Enabling Embedded C++ Mode (−−embedded_cpp 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
- Run-time 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 MSP430 compiler nevertheless allows these features under embedded C++ because the C++ run-time support library makes use of them. Furthermore, these features impose no run-time penalty.

5.13 Compiler Limits

Due to the variety of host systems supported by the MSP430 C/C++ compiler and the limitations of some of these systems, the compiler may not be able to successfully compile source files that are excessively large or complex. In general, exceeding such a system limit prevents continued compilation, so the compiler aborts immediately after printing the error message. Simplify the program to avoid exceeding a system limit.

Some systems do not allow filenames longer than 500 characters. Make sure your filenames are shorter than 500.

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

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

Chapter 6

Run-Time Environment

This chapter describes the MSP430 C/C++ run-time environment. To ensure successful execution of C/C++ programs, it is critical that all run-time code maintain this environment. It is also important to follow the guidelines in this chapter if you write assembly language functions that interface with C/C++ code.

Topic **Page**

6.1 Memory Model

The MSP430 treats memory as a single linear 16-bit-wide address space that is partitioned into sections. Each section contains a continuous region of memory. Each block of code or data generated by a C/C++ program is placed into one of these sections by the link step.

When the −−silicon_version=mspx option is specified, the compiler supports a large code memory model while generating code for MSP430x devices. The size of function pointers is 20-bits. As for MSP430 devices, the size of data pointers is 16-bits. A large code memory model allows unrestricted placement of programs in the 20-bit address space with the exception of interrupt service routines (see section 6.6.5, *Other Interrupt Information*, on page 6-24). However, data is still required to be placed into the low 64K of memory. Accessing data located in memory above 64K requires interfacing to assembly routines from C.

Modules assembled/compiled for 16-bit MSP devices are not compatible with modules that are assembled/compiled for 20-bit MSPx devices. The link step generates errors if an attempt is made to combine incompatible object files.

Note: The Link Step Defines the Memory Map

The link step, 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 link step to allocate code and data into the appropriate memory spaces.

For example, you can use the link step 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).

6.1.1 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 *MSP430 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* and the *.pinit section* contain tables for initializing variables and constants.
  - The *.const section* contains string constants, switch tables, and data defined with the C/C++ qualifier *const* (provided the constant is not also defined as *volatile*).
  - The *.text section* contains executable code as well as string literals and compiler-generated constants.
- *Uninitialized sections* reserve space in memory (usually RAM). A program can use this space at run time for creating and storing variables. The compiler creates the following uninitialized sections:
  - The *.bss section* reserves space for global and static variables. At boot or load time, the C/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 C/C++ software stack.
  - The *.sysmem section* reserves space for dynamic memory allocation. This space is used by dynamic memory allocation routines, such as malloc, calloc, realloc, or new. If a C/C++ program does not use these functions, the compiler does not create the .sysmem section.

See section 5.8.4, *The DATA_SECTION Pragma*, on page 5-16 if you want to allocate space for global and static variables in a named section other than .bss.

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

The link step takes the individual sections from different modules and combines sections that have the same name. The resulting 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	.pinit	ROM or RAM
.cinit	ROM or RAM	.stack	RAM
.const	ROM or RAM	.sysmem	RAM
.data	ROM or RAM	.text	ROM or RAM

You can use the SECTIONS directive in the link step command file to customize the section-allocation process. For more information about allocating sections into memory, see the link step description chapter in the *MSP430 Assembly Language Tools User's Guide*.

6.1.2 C/C++ Software Stack

The C/C++ compiler uses a stack to perform the following tasks:

- ❑ Allocate local variables
- ❑ Pass arguments to functions
- ❑ Save register contents

The run-time stack grows down from high addresses to lower addresses. The compiler uses the SP (R1) register to manage the stack. The *stack pointer* is initialized to point to the top of the stack (the highest address used).

The link step sets the stack size, creates a global symbol, _ _STACK_SIZE, and assigns it a value equal to the size of the stack in bytes. The default stack size is 128 bytes. You can change the size of the stack at link time by using the −−stack_size option with the link step command. For more information on the stack option, see section 4.2, *Link Step Options*, on page 4-5.

Note: Save-On-Entry Registers and C/C+ Stack Size

Since register sizes increase for MSP430X devices (specified with −−silicon_version=mspx), saving and restoring save-on-entry registers requires 32-bits of stack space for each register saved on the stack. When you are porting code originally written for 16-bit MSP430 devices, you may need to increase the C stack size from previous values.

At system initialization, SP is set to a designated address for the top of the stack. This address is the first location past the end of the .stack section. Since the position of the stack depends on where the .stack section is allocated, the actual address of the stack is determined at link time.

The compiler uses SP to mark the top of the stack. The C/C++ environment automatically decrements SP at the entry to a function to reserve all the space necessary for the execution of that function. The stack pointer is incremented at the exit of the function to restore the stack to the state before the function was entered. If you interface assembly language routines to C/C++ programs, be sure to restore the stack pointer to the same state it was in before the function was entered. (For more information about using the stack pointer, see section 6.3, *Register Conventions*, on page 6-11; for more information about the stack, see section 6.4, *Function Structure and Calling Conventions*, on page 6-13.)

Note: Stack Overflow

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

6.1.3 Dynamic Memory Allocation

The run-time-support library supplied with the compiler contains several functions (such as malloc, calloc, and realloc) that allow you to dynamically allocate memory for variables at run time.

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 link step command. The link step 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 128 bytes. For more information on the −−heap_size option, see section 4.2, *Link Step Options*, on page 4-5.

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));
```

6.1.4 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 run time. You can specify this to the link step by using the −−ram_model link step option. For more information, see section 6.9, *System Initialization*, on page 6-30.

6.2 Object Representation

This section explains how various data objects are sized, aligned, and accessed.

6.2.1 Data Type Storage

Table 6−2 lists register and memory storage for various data types:

Table 6−2. Data Representation in Registers and Memory

Data Type	Register Storage	Memory Storage
char, signed char	Bits 0−7 of register†	8 bits
unsigned char, bool	Bits 0−7 of register	8 bits
short, signed short	Bits 0−15 of register†	16 bits, word (16 bits) aligned
unsigned short, wchar_t	Bits 0−15 of register	16 bits, word (16 bits) aligned
int, signed int	Bits 0−15 of register†	16 bits, word (16 bits) aligned
unsigned int	Bits 0−15 of register	16 bits, word (16 bits) aligned
enum	Bits 0−15 of register†	16 bits, word (16 bits) aligned
long, signed long	Register pair	32 bits, word (16 bits) aligned
unsigned long	Register pair	32 bits, word (16 bits) aligned
float	Register pair	32 bits, word (16 bits) aligned
double	Register pair	32 bits, word (16 bits) aligned
long double	Register pair	32 bits, word (16 bits) aligned
struct	Members stored as their individual types require	Members stored as their individual types require; aligned according to the member with the most restrictive alignment requirement
array	Members stored as their individual types require	Members stored as their individual types require; word aligned. All arrays inside a structure are aligned according to the type of each element in the array.
pointer to data member	Bits 0−15 of register	16 bits, word (16 bits) aligned
MSP pointer to function	Bits 0−15 of register	16 bits, word (16 bits) aligned
MSPX‡ pointer to function	Bits 0−20 of register	32 bits, word (16 bits) aligned

† Negative values are sign-extended to bit 15.
‡ MSP430x (MSPX) is specified with the −−silicon_version=mspx option.

6.2.1.1 Pointer to Member Function Data Type

Pointer to member function objects are stored as a structure with three members, and the layout is equivalent to:

```
struct {
   short int d;
   short int i;
union {
   void (f) ( );
   long o;
   }
}
```

where, *d* is the offset to be added to the beginning of the class object for the pointer, and *i* is the index into the virtual function table offset by one so that the NULL pointer can be represented. If the value of *i* is −1 the function is non-virtual. In a non-virtual function (where i is −1), *f* is the pointer to the member function. Other-wise, *o* is the offset to the virtual function table pointer within the class object.

6.2.1.2 Structure and Array Alignment

Structures are aligned according to the member with the most restrictive alignment requirement. Structures do not contain padding after the last member. Arrays are always word aligned. Elements of arrays are stored in the same manner as if they were individual objects.

6.2.1.3 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 1-word (16-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 either word or byte boundaries. Fields are allocated as many bits as requested. Adjacent fields are packed into adjacent bits of a word, but they do not overlap words; if a field would overlap into the next word, the entire field is placed into the next word.

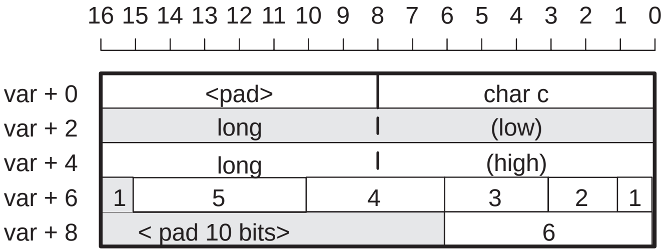
Fields are packed as they are encountered;the least significant bits of the structure word are filled first. Example 6−1 shows the C code definition and memory layout of var.

Example 6−1. Field/Structure Alignment of Var

(a) C code definition of var

```
struct example {
    char c;
    long l;
    int bf1:1;
    int bf2:2;
    int bf3:3;
    int bf4:4;
    int bf5:5;
    int bf6:6;
};
```

(b) Memory layout of var



6.2.2 Character String Constants

In C/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 6.9, *System Initialization*, on page 6-30.

❑ In an expression. For example:

```
strcpy (s, ”abc”);
```

When a string is used in an expression, the string itself is defined in the .text 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:

```
      .text
      .align   4
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 0 and is increased by 1 for each string defined. All strings used in a source module are defined at the end of the compiled assembly language module.

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

If the same string is used more than once within a source module, the 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 6−3 shows the types of registers affected by these conventions, and Table 6−4 shows how the compiler uses registers and whether their values are preserved across calls. For information about how values are preserved across calls, see section 6.4, *Function Structure and Calling Conventions*.

Table 6−3. How Register Types Are Affected by the Conventions

Register Type	Description
Argument register	Passes arguments during a function call
Return register	Holds the return value from a function call
Expression register	Holds a value
Argument pointer	Used as a base value from which a function’s parameters (incoming arguments) are accessed
Stack pointer	Holds the address of the top of the software stack
Program counter	Contains the current address of code being executed

Table 6−4. Register Usage and Preservation Conventions

Register	Alias	Usage	Preserved by Function†
R0	PC	Program counter	N/A
R1	SP	Stack pointer	N/A‡
R2	SR	Status register	N/A
R3		Constant generator	N/A
R4−R10		Expression register	Child
R11		Expression register	Parent
R12		Expression register, argument pointer, return register	Parent
R13		Expression register, argument pointer, return register	Parent
R14		Expression register, argument pointer	Parent
R15		Expression register, argument pointer	Parent

† The parent function refers to the function making the function call. The child function refers to the function being called.
‡ The SP is preserved by the convention that everything pushed on the stack is popped off before returning.

6.4 Function Structure and Calling Conventions

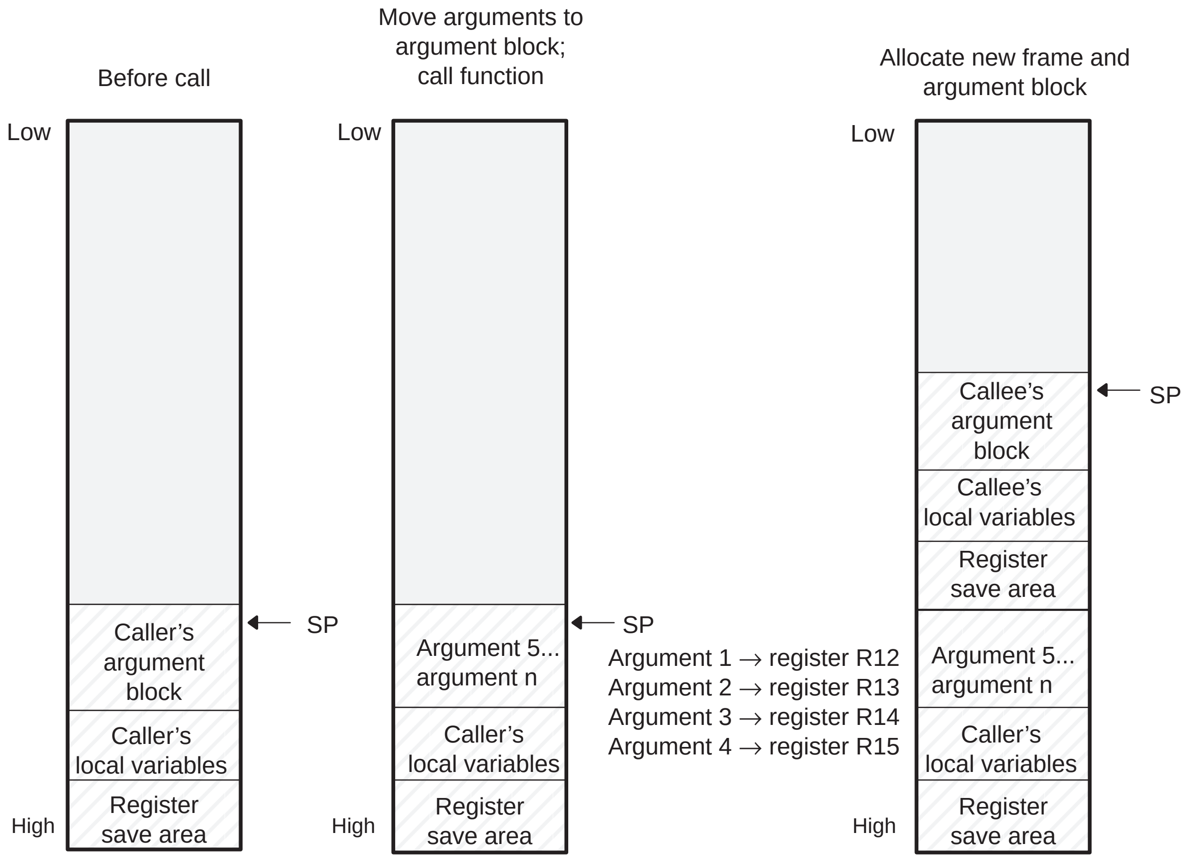
The C/C++ compiler imposes a strict set of rules on function calls. Except for special run-time-support functions, any function that calls or is called by a C/C++ function must follow these rules. Failure to adhere to these rules can disrupt the C/C++ environment and cause a program to fail.

The following sections use this terminology to describe the function-calling conventions of the C/C++ compiler:

- **Argument block.** The part of the local frame used to pass arguments to other functions. Arguments 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.
- **Register save area.** The part of the local frame that is used to save the registers when the program calls the function and restore them when the program exits the function.
- **Save-on-call registers.** Registers R11−R15. The called function does not preserve the values in these registers; therefore, the calling function must save them if their values need to be preserved.
- **Save-on-entry registers.** Registers R4−R10. It is the called function's responsibility to preserve the values in these registers. If the called function modifies these registers, it saves them when it gains control and preserves them when it returns control to the calling function.

Figure 6−1 illustrates a typical function call. In this example, arguments are passed to the function, and the function uses local variables and calls another function. The first four arguments are passed to registers R12−R15. This example also shows allocation of a local frame and argument block for the called function. Functions that have no local variables and do not require an argument block do not allocate a local frame.

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



Legend: AP: argument pointer
SP: stack pointer

6.4.1 How a Function Makes a Call

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

1) The caller is responsible for preserving any save-on-call registers across the call that are live across the call. (The save-on-call registers are R11−R15.)

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

3) The caller places the first arguments in registers R12−R15, in that order. The caller moves the remaining arguments to the argument block in reverse order, placing the leftmost remaining argument at the lowest address. Thus, the leftmost remaining argument is placed at the top of the stack.

4) The caller calls the function.

6.4.2 How a Called Function Responds

A called function performs the following tasks:

1) If the function is declared with an ellipsis, it can be called with a variable number of arguments. The called function pushes these arguments on the stack if they meet both of these criteria:

   - The argument includes or follows the last explicitly declared argument.
   - The argument is passed in a register.

2) The called function pushes register values of all the registers that are modified by the function and that must be preserved upon exit of the function onto the stack. Normally, these registers are the save-on-entry registers (R4−R10) if the function contains calls. If the function is an interrupt, additional registers may need to be preserved. For more information, see section 6.6, *Interrupt Handling*, on page 6-23.

3) The called function allocates memory for the local variables and argument block by subtracting a constant from the SP. This constant is computed with the following formula:

   size of all local variables + max = constant

   For the above, the *max* is the size of all the parameters placed in the argument block for each call.

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

5) If the called function returns a value, it places the value in R12 (or R12 and R13 for floating-point or long values).

6) If the called function returns a structure, it copies the structure to the memory block that the first argument, R12, points to. If the caller does not use the return value, R12 is set to 0. This directs the called function not to copy the return structure.

   In this way, the caller can be smart about telling the called function where to return the structure. For example, in the statement:

   ```
   s = f( )
   ```

   where s is a structure and f is a function that returns a structure, the caller can simply pass the address of s as the first argument and call f. Function f then copies the return structure directly into s, performing the assignment automatically.

   You must be careful to properly declare functions that return structures, both at the point where they are called (so the caller properly sets up the first argument) and where they are defined (so the function knows to copy the result).

7) The called function deallocates the frame and argument block by adding the constant computed in step 3.

8) The called function restores all registers saved in step 2.

9) The called function (_f) returns.

The following example is typical of how a called function responds to a call:

```
func:                          ; Called function entry point
        PUSH.W    r10
        PUSH.W    r9           ; Save SOE registers
        SUB.W     #2,SP        ; Allocate the frame
        :
        :                      ; Body of function
        :
        ADD.W     #2,SP        ; Deallocate the frame
        POP       r9           ; Restore SOE registers
        POP       r10
        RET                    ; Return
```

6.4.3 Accessing Arguments and Local Variables

A function accesses its local nonregister variables indirectly through the stack pointer (SP) and its stack arguments. The SP always points to the top of the stack (points to the most recently pushed value).

Since the stack grows toward smaller addresses, the local data on the stack for the C/C++ function is accessed with a positive offset from the SP register.

6.5 Interfacing C/C++ With Assembly Language

These are ways to use assembly language in conjunction with C/C++ code:

- Use separate modules of assembled code and link them with compiled C/C++ modules (see section 6.5.1). This is the most versatile method.
- Use assembly language variables and constants in C/C++ source (see section 6.5.2 on page 6-20).
- Use inline assembly language embedded directly in the C/C++ source (see section 6.5.4 on page 6-22).

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*, on page 6-11 and the calling conventions defined in section 6.4, *Function Structure and Calling Conventions,* on page 6-13. 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 (R4−R10)
  - Stack pointer

  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.

- Interrupt routines must save *all* the registers they use. For more information see section 6.6, *Interrupt Handling*, on page 6-23.
- When you call a C/C++ function from an assembly language program, load the designated register with arguments and push the remaining arguments on the stack as described in section 6.4.1, *How a Function Makes a Call*, on page 6-15.

  Remember that a function can alter any register not designated as being preserved without having to restore it. If the contents of any of these registers must be preserved across the call, you must explicitly save them.

- Functions must return values correctly according to their C/C++ declarations. Long values are returned in R12 and R13, and structures are returned as described in step 2 of section 6.4.1, *How a Function Makes a Call*, on page 6-15. Any other values are returned in R12.

- 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 assigns a linkname to all external objects. See section 5.10, *Generating Linknames*, for details. This means that when writing assembly language code, you must use the same linknames as those assigned by the compiler.

  For identifiers to be used only in an assembly language module ensure that they do not conflict with a C/C++ identifier. Use a $ or a consistent prefix naming convention to create a different namespace for assembly identifiers.

- 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 link step 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 link step resolves.

- Any assembly routines that interface with MSP430x C programs are required to conform to the large code model:

  - Use CALLA/RETA instead of CALL/RET
  - Use PUSHM.A/POPM.A to save and restore any used save-on-entry registers. The entire 20-bit register must be saved/restored.
  - Manipulation of function pointers requires 20-bit operations (OP.A)

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

Example 6−2. An Assembly Language Function

(a) C++ program

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


void main()
{
   int i = 5;

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

(b) Assembly language program

```
      .global asmfunc
      .global gvar

asmfunc:
      MOV   &gvar,R11
      ADD   R11,R12
      RET
```

In the C++ program in Example 6−2, the extern “C” declaration tells the compiler to use C naming conventions (i.e., no name mangling). When the link step resolves the .global asmfunc reference, the corresponding definition in the assembly file will match.

The parameter i is passed in R12, and the result is returned in R12.

6.5.2 Accessing Assembly Language Variables From 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) Use the appropriate linkname in assembly language.
4) In C, declare the variable as extern and access it normally.

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

Example 6−3. Accessing an Assembly Language Variable From C

(a) Assembly language program

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

    .bss      var,4,4   ; Define the variable
    .global   var       ; Declare it as external
```

(b) C program

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

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 link step command file using a link step 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 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 link step) 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 6−4.

Example 6−4. 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 6−4, int is used. You can reference link step-defined symbols in a similar manner.

6.5.3 Sharing C/C++ Header Files With Assembly Source

You can use the .cdecls assembler directive to share C headers containing declarations and prototypes between C and assembly code. Any legal C/C++ can be used in a .cdecls block and the C/C++ declarations will cause suitable assembly to be generated automatically, allowing you to reference the C/C++ constructs in assembly code. For more information , see the *Sharing C/C++ Header Files in Assembly Source* chapter of the *MSP430 Assembly Language Tools User's Guide.*

6.5.4 Using Inline Assembly Language

Within a C/C++ program, you can use the asm statement to insert a single line of assembly language into the assembly language file 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 5.7, *The asm Statement*, on page 5-12.

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.

6.6 Interrupt Handling

As long as you follow the guidelines in this section, you can interrupt and return to C/C++ code 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, you must handle any required enabling or masking of interrupts. Such operations have no effect on the environment and can be easily incorporated with asm statements.

6.6.1 Saving Registers During Interrupts

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 functions called by the routine. Register preservation must be explicitly handled by the interrupt routine.

6.6.2 Using C/C++ Interrupt Routines

A C/C++ interrupt routine is like any other C/C++ function in that it can have local variables and register variables. An interrupt routine must be declared with no arguments and must return void . For example:

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

If a C/C++ interrupt routine does not call any other functions, only those registers that the interrupt handler uses are saved and restored. However, if a C/C++ interrupt routine *does* call other functions, these functions can modify unknown registers that the interrupt handler does not use. For this reason, the routine saves all the save-on-call registers if any other functions are called. (This excludes banked registers.) Do not call interrupt handling functions directly.

Interrupts can be handled *directly* with C/C++ functions by using the interrupt pragma or the interrupt keyword. For information, see section 5.8.11, *The INTERRUPT Pragma*, on page 5-20, and section 5.4.2, *The interrupt Keyword*, on page 5-8.

6.6.3 Using Assembly Language Interrupt Routines That Call C/C++ Functions

You can handle interrupts with assembly language code as long as you follow the same register conventions the compiler does. Like all assembly functions, interrupt routines can use the stack, access global C/C++ variables, and call C/C++ functions normally. When calling C/C++ functions, be sure that all save-on-call registers are preserved before the call because the C/C++ function can modify any of these registers. You do not need to save save-on-entry registers because they are preserved by the called C/C++ function.

6.6.4 Interrupt Vectors

The interrupt vectors for the MSP430 and MSP430x devices are 16 bits. Therefore, interrupt service routines (ISR's) must be placed into the low 64K of memory. Convenience macros are provided in the MSP430x device headers file to declare interrupts to ensure 16-bit placement when linking.

Alternatively, use the CODE_SECTIONS pragma to place the code for ISRs into sections separate from the default .text sections. Use the link step command file and the SECTIONS directive to ensure the code sections associated with ISRs are placed into low memory.

6.6.5 Other Interrupt Information

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

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

- It is your responsibility to handle any special masking of interrupts.
- A C/C++ interrupt routine cannot be called explicitly.
- In a system reset interrupt, such as c_int00, you cannot assume that the run-time environment is set up; therefore, you *cannot allocate local variables*, and you *cannot save any information on the run-time stack*.
- In assembly language, remember to precede the name of a C/C++ interrupt with the appropriate linkname.

6.7 Intrinsic Run-Time-Support Arithmetic and Conversion Routines

The intrinsic run-time-support library contains a number of assembly language routines that provide arithmetic and conversion capability for C/C++ operations that the 32-bit and 16-bit instruction sets do not provide. These routines include integer division, , integer multiply, integer modulus, and floating-point operations.

The source files for these functions are in the rts.src library. The source code has comments that describe the operation of the functions. You can extract, inspect, and modify any of these functions.

Note that the run-time-support routines follow the register calling conventions described in section 6.3, *Register Conventions*, on page 6-11, except for the following:

- **Integer modulus routines.** The return value for 8 and 16-bit integer modulus is placed in R14. The return value for 32-bit integer modulus is placed in R14/R15.
- **Floating compare**. The result of a floating point compare is represented by the status register.

6.8 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:

```
short state;
    :
state = _get_SR_register();
```

No declaration of the intrinsic functions is necessary.

6.8.1 MSP430 Intrinsics

Table 6−4 lists all of the intrinsic operators in the MSP430 C/C++ compiler. A function-like prototype is presented for each intrinsic that shows the expected type for each parameter. If the argument type does not match the parameter, type conversions are performed on the argument.

For more information on the resulting assembly language mnemonics, see the *MSP430x1xx Family User's Guide*, the *MSP430x3xx Family User's Guide*, and the *MSP430x4xx Family User's Guide*.

Table 6−5. MSP430 Intrinsics

Intrinsic		**Generated Assembly**
void	_nop(void)	NOP
void	_enable_interrupts(void)	EINT
void	_disable_interrupts(void)	DINT
unsigned short	_swap_bytes(short a)	MOV *a*, *dst* SWPB *dst*
unsigned short	_bic_SR_register(unsigned short *mask*)	BIC *mask*, SR
unsigned short	_bic_SR_register_on_exit(unsigned short *mask*)	BIC *mask*, saved_SR
unsigned short	_bis_SR_register(unsigned short *mask*)	BIS *mask*, SR
unsigned short	_bis_SR_register_on_exit(unsigned short *mask*)	BIS *mask*, saved_SR
unsigned short	_get_SR_register(void)	MOV SR, *dst*
unsigned short	_get_SR_register_on_exit(void)	MOV saved_SR, *dst*

Table 6−5. MSP430 Intrinsics (continued)

Intrinsic		Generated Assembly
void	_never_executed( )	See section 6.8.2.
unsigned short	_bcd_add_short(unsigned short op1, unsigned short op2)	MOV *op1*, *dst* CLRC DADD *op2*, *dst*

unsigned long	_bcd_add_long(unsigned long, unsigned long)	MOV *op1_low*, *dst_low*
		MOV *op1_hi*, *dst_hi*
		CLRC
		DADD *op2_low*, *dst_low*
		DADD *op2_hi*, *dst_hi*

6.8.2 The _never_executed Intrinsic

The MSP430 C/C++ Compiler supports a _never_executed( ) intrinsic that can be used to assert that a default label in a switch block is never executed. If you assert that a default label is never executed the compiler can generate more efficient code based on the values specified in the case labels within a switch block.

6.8.2.1 Using _never_executed With a Vector Generator

The _never_executed( ) intrinsic is specifically useful for testing the values of an MSP430 interrupt vector generator such as the vector generator for Timer A (TAIV). MSP430 vector generator values are mapped to an interrupt source and are characterized in that they fall within a specific range and can only take on even values. A common way to handle a particular interrupt source represented in a vector generator is to use a switch statement. However, a compiler is constrained by the C language in that it can make no assumptions about what values a switch expression may have. The compiler will have to generate code to handle every possible value, which leads to what would appear to be inefficient code.

The _never_executed( ) intrinsic can be used to assert to the compiler that a switch expression can only take on values represented by the case labels within a switch block. Having this assertion, the compiler can avoid generating test code for handling values not specified by the switch case labels. Having this assertion is specifically suited for handling values that characterize a vector generator.

Example 6−5 illustrates a switch block that handles the values of the Timer B (TBIV) vector generator.

Example 6−5. TBIV Vector Generator

```
__interrupt void Timer_B1 (void)
{
 switch( TBIV )
 {
   case  0: break; /* Do nothing */
   case  2: TBCCR1 += 255;
            state +=1;
            break;
   case  4: TBCCR0 = 255−1;
            TBCCR1 = 255−96;
            state =200;
            break;
   case  6:
   case  8:
   case 10:
   case 12:
   case 14:
   default:  _never_executed();
 }
}
```

In Example 6−5 using the _never_executed( ) intrinsic asserts that the value of TBIV can only take on the values specified by the case labels, namely the even values from 0 to 14. Normally, the compiler would have to generate code to handle any value which would result in extra range checks. Instead, for this example, the compiler will generate a switch table where the value of TBIV is simply added to the PC to jump to the appropriate code block handling each value represented by the case labels.

6.8.2.2 Using _never_executed With General Switch Expressions

Using the _never_executed( ) intrinsic at the default label can also improve the generated switch code for more general switch expressions that do not involve vector generator type values.

Example 6−6. General Switch Statement

```
switch( val)
{
  case  0:
  case  5: action(a); break;

  case 14: action(b); break;

  default:  _never_executed();
}
```

Normally, for the switch expression values 0 and 5, the compiler generates code to test for both 0 and 5 since the compiler must handle the possible values 1−4. The _never_executed( ) intrinsic in Example 6−6 asserts that *val* can not take on the values 1−4 and therefore the compiler only needs to generate a single test (val < 6) to handle both case labels.

Additionally, using the _never_executed( ) intrinsic results in the assertion that if val is not 0 or 5 then it has to be 14 and the compiler has no need to generate code to test for val == 14.

The _never_executed( ) intrinsic is only defined when specified as the single statement following a default case label. The compiler ignores the use of the intrinsic in any other context.

6.9 System Initialization

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

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 −−rom_model or −−ram_model link step function option and include the run-time library as one of the link step input files.

When C/C++ programs are linked, the link step 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) Reserves space for the user mode run-time stack and sets up the initial value of the stack pointer (SP).
2) Initializes global variables by copying the data from the initialization tables in the .cinit section to the storage allocated for the variables in the .bss section. If initializing variables at load time (−−ram_model option), a loader performs this step before the program runs (it is not performed by the boot routine). For more information, see section 6.9.3, *Automatic Initialization of Variables*.
3) Executes the global constructors found in the .pinit section. For more information, see section 6.9.4, *Global Constructors*.
4) 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.

6.9.1 System Pre-Initialization

The _c_int00( ) initialization routine also provides a mechanism for an application to perform the MSP430 setup (set I/O registers, enable/disable timers, etc.) before the C/C++ environment is initialized.

Before calling the routine that initializes C/C++ global data and calls any C++ constructors, the boot routine makes a call to the function _system_pre_init( ). A developer can implement a customized version of _system_pre_init( ) to perform any application-specific initialization before proceeding with C/C++ environment setup. In addition, the default C/C++ data initialization can be bypassed if _system_pre_init( ) returns a 0. By default, _system_pre_init( ) should return a non−zero value.

In order to perform application-specific initializations, you can create a customized version of _system_pre_init( ) and add it to the application project. The customized version will replace the default definition included in the run-time library if it is linked in before the run-time library.

The default stubbed version of _system_pre_init( ) is included with the run-time library. It is located in the file pre_init.c and is included in the run-time source library (rts.src). The archiver utility (ar430) can be used to extract pre_init.c from the source library.

6.9.2 Run-Time Stack

The run-time stack is allocated in a single continuous block of memory and grows down from high addresses to lower addresses. The SP points to the top of the stack.

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

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

The C/C++ boot routine shipped with the compiler sets up the user mode run-time stack. If your program uses a run-time stack when it is in other operating modes, you must also allocate space and set up the run-time stack corresponding to those modes.

6.9.3 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 link step 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.

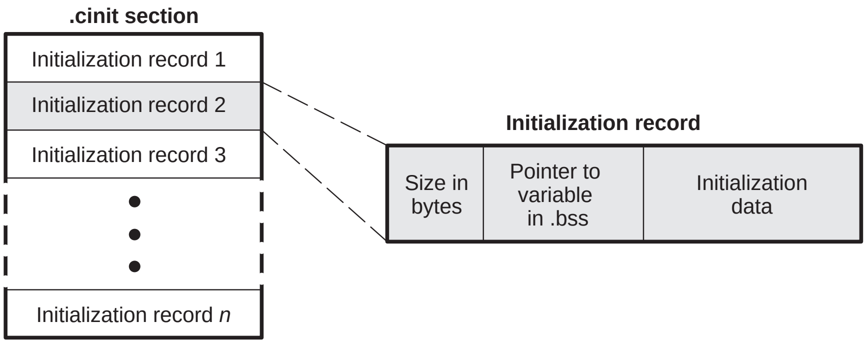
6.9.4 Global Constructors

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 link step 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.

6.9.5 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 6−2 shows the format of the .cinit section and the initialization records.

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



An initialization record contains the following information:

- The first field contains the size in bytes of the initialization data. The width of this field is one word (16 bits).
- The second field contains the starting address of the area where the initialization data must be copied. The width of this field is one word.
- The third field contains the data that is copied to initialize the variable. The width of this field is variable.

The .cinit section contains an initialization record for each variable that is initialized. Example 6−7 shows two initialized variables defined in C. Example 6−7 (b) shows the corresponding initialization table.

Example 6−7. Initialization Variables and Initialization Table

(a) Initialized variables defined in C

```
int    i = 23;
int    a[5] = { 1, 2, 3, 4, 5 };
```

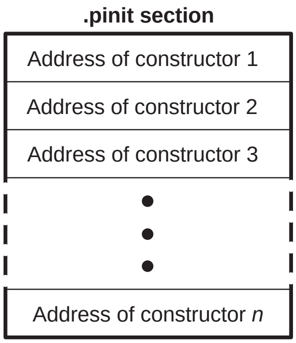
(b) Initialized information for variables defined in (a)

```
    .sect  ".cinit"
    .align 2
    .field    2,16
    .field    i+0,16
    .field    23,16         ; i @ 0

    .sect  ".cinit"
    .align 2
    .field    $C$IR_1,16
    .field    a+0,16
    .field    1,16          ; a[0] @ 0
    .field    2,16          ; a[1] @ 16
    .field    3,16          ; a[2] @ 32
    .field    4,16          ; a[3] @ 48
    .field    5,16          ; a[4] @ 64
$C$IR_1:  .set   10

    .global   i
    .bss   i,2,2
    .global   a
    .bss   a,10,2
```

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.

The table in the .pinit section simply consists of a list of addresses of constructors to be called (see Figure 6−3). The constructors appear in the table in the order they must be executed.

Figure 6−3. Format of the .pinit Section



When you use the −−rom_model or −−ram_model option, the link step combines the .cinit sections from all the C/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.

Likewise, the −−rom_model or −−ram_model link step option causes the link step to combine all of the .pinit sections from all C/C++ modules and append 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.

The const-qualified variables are initialized differently; see section 5.4.1, *The const Keyword*, on page 5-7.

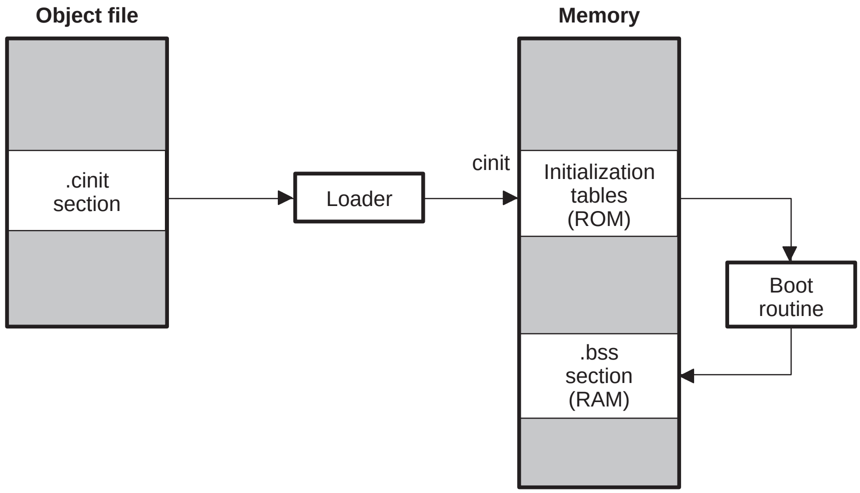
6.9.6 Autoinitialization of Variables at Run Time

Autoinitializing variables at run time is the default model for autoinitialization. To use this method, invoke the link step with the −−rom_model option.

Using this method, the .cinit section is loaded into memory along with all the other initialized sections, and global variables are initialized at run time. The link step 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 6−4 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into ROM.

Figure 6−4. Autoinitialization at Run Time



6.9.7 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 link step with the −−ram_model option.

When you use the −−ram_model link step option, the link step 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 link step also sets the cinit symbol to −1 (normally, cinit points to the beginning of the initialization tables). This indicates to the boot routine that the initialization tables are not present in memory; accordingly, no run-time initialization is performed at boot time.

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

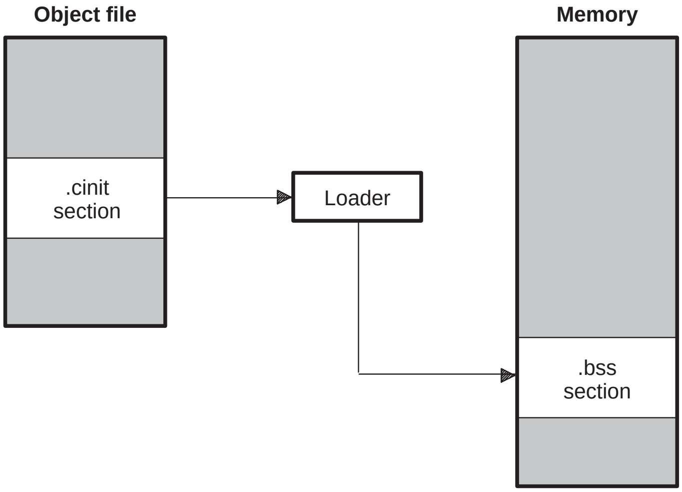
❑ Detect the presence of the .cinit section in the object file

❑ Determine that STYP_COPY is set in the .cinit section header, so that it knows not to copy the .cinit section into memory

❑ Understand the format of the initialization tables

Figure 6−5 illustrates the RAM model of autoinitialization.

Figure 6−5. Autoinitialization at Load Time



Whether or not you use the −−rom_model or −−ram_model link step option, the .pinit section is always loaded and processed at run time.

6.10 Compiling for 20-Bit MSPX Devices

The MSP430 tools support compiling and linking code for MSP430 and MSP430X (MSPX) devices. See the following for more information on options and topics that apply to compiling for the MSPX devices:

- ❑ Use the −−silicon_version=mspx option to compile for MSPX devices. See page 2-17.
- ❑ Function pointers are 20-bits. See Table 5−1 on page 5-6 and Table 6−2 on page 6-7.
- ❑ The compiler supports a large code memory model while generating code for MSP430x devices. See section 6.1, *Memory Model*, on page 6-2.
- ❑ Any assembly routines that interface with MSP430x C programs must fit the large code model. See 6.5.1, *Using Assembly Language Modules with C/C++ Code*, on page 6-17.
- ❑ Interrupt service routines must be placed into low memory. See section 6.6.4, *Interupt Vectors*, on page 6-24.
- ❑ Link with the rts430x.lib or rts430x_eh.lib run-time-support library. See section 8.2.1, *The Base Option Sets for Building the Libraries*, on page 8-3.

Chapter 7

Run-Time-Support Functions

Some of the tasks that a C/C++ program performs (such as I/O, dynamic memory allocation, string operations, and trigonometric functions) are not part of the C/C++ language itself. However, the ANSI C standard defines a set of run-time-support functions that perform these tasks. The MSP430 C/C++ compiler implements the complete ANSI C standard library except for these facilities that handle exception conditions and locale issues (C properties that depend on local language, nationality, or culture). Using the ANSI standard library ensures a consistent set of functions that provide for greater portability.

In addition to the ANSI-specified functions, the MSP430 run-time-support library includes routines that give you processor-specific commands and direct C language I/O requests.

A library-build process is included with the code generation tools that lets you create customized run-time-support libraries. For information about using the library-build process, see Chapter 8.

Topic **Page**

7.1 Libraries

When you link your program, you must specify an object library as one of the link step input files so that references to the I/O and run-time-support functions can be resolved. For 16-bit MSP devices, link with rts430.lib. For 20-bit MSPX devices, link with rts430x.lib.

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

When a library is linked, the link step includes only those library members required to resolve undefined references. For more information about linking, see the link step description chapter of the *MSP430 Assembly Language Tools User's Guide*.

7.1.1 Nonstandard Header Files in rtssrc.zip

The *rtssrc.zip* self-processing zip file contains these non-ANSI include files that are used to build the library:

- The *values.h* file contains the definitions necessary for recompiling the trigonometric and transcendental math functions. If necessary, you can customize the functions in values.h.
- The *file.h* file includes macros and definitions used for low-level I/O functions.
- The *format.h* file includes structures and macros used in printf and scanf.
- The *rtti.h* file includes internal function prototypes necessary to implement run-time type identification.
- The *vtbl.h* file contains the definition of a class's virtual function table format.

7.1.2 Modifying a Library Function

You can inspect or modify library functions by unzipping the source file (rtssrc.zip), changing the specific function file, and rebuiliding the library with the self-processing zip file. See Chapter 8.

7.1.3 Building a Library With Different Options

See Chapter 8.

7.2 The C I/O Functions

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

To use the I/O functions:

- Include the header file stdio.h for each module that references a function.
- Allow for 320 bytes of heap space for each I/O stream used in your program. A stream is a source or destination of data that is associated with a peripheral, such as a terminal or keyboard. Streams are buffered using dynamically allocated memory that is taken from the heap. More heap space may be required to support programs that use additional amounts of dynamically allocated memory (calls to malloc( )). To set the heap size, use the −heap option when linking. See section 4.2, *Link Step Options,* on page 4-5, for more information about the −heap option.

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

```
#include <stdio.h>

main()
{
 FILE *fid;

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

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

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

```
cl430 main.c --run_linker --library=rts430.lib --output_file=main.out
```

Executing main.out under the MSP430 debugger on a host accomplishes the following tasks:

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

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

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.

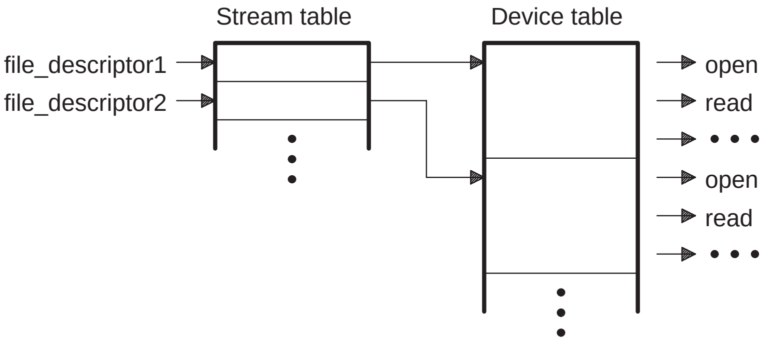
The high-level functions are the standard C library of stream I/O routines (printf, scanf, fopen, getchar, etc.). These routines map an I/O request to one or more of the I/O commands that are handled by the low-level shell.

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

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

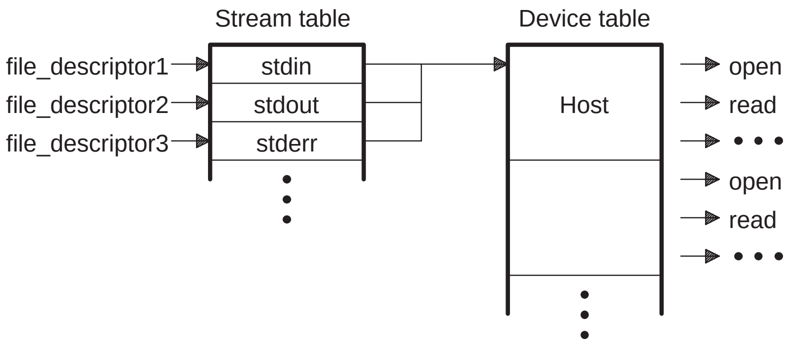
The data structures interact as shown in Figure 7−1.

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 in section 7.2.2, *Adding a Device for C I/O*, on page 7-13. 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.

add_device — *Add Device to Device Table*

Syntax

#include <file.h>
int add_device(char **name***,**
unsigned *flags***,**
int (*dopen**)( ),** *parameters*,
int (*dclose**)( ),** *parameters*,
int (*dread**)( ),** *parameters*,
int (*dwrite**)( ),** *parameters*,
fpos_t (*dlseek**)( ),** *parameters*,
int (*dunlink**)( ),** *parameters*,
int (*drename**)( ),** *parameters***);**

Defined in

lowlev.c in rts.src

Description

The add_device function adds a device record to the device table, allowing that device to be used for I/O from C/C++. The first entry in the device table is predefined to be the host device on which the debugger is running. The function add_device( ) finds the first empty position in the device table and initializes the fields of the structure that represent the device added.

To open a stream on a newly-added device, use fopen( ) with a string of the format *devicename***:***filename* as the first argument.

- The *name* is a character string denoting the device name.
- The *flags* are device characteristics. The flags are as follows:

  _SSA Denotes that the device supports only one open stream at a time
  _MSA Denotes that the device supports multiple open streams

  More flags can be added by defining them in stdio.h/cstdio.

- The dopen, dclose, dread, dwrite, dlseek, dunlink, and drename specifiers are function pointers to the device drivers that are called by the low-level functions to perform I/O on the specified device. You must declare these functions with the interface specified in section 7.2.1, *Overview of Low-Level I/O Implementation*, on page 7-4. The device drivers for the host that the debugger is run on are included in the C/C++ I/O library.

Return Value

The function returns one of the following values:

0	if successful
−1	if fails

Example This example 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);
}
```

close — *Close File or Device for I/O*

Syntax for C

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

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

Syntax for C++

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

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 function returns one of the following values:

0	if successful
−1	if fails

lseek — *Set File Position Indicator*

Syntax for C

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

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

Syntax for C++

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

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

Description

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

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

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

Return Value

The function returns one of the following values:

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

open — *Open File or Device for I/O*

Syntax for C

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

int open(const char ***_path_, unsigned** _flags_**, int** _file_descriptor_**);**

Syntax for C++

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

int std::open(const char ***_path_, unsigned** _flags_**, int** _file_descriptor_**);**

Description

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

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

- The *flags* are attributes that specify how the device or file is manipulated. The flags are specified using the following symbols:

```
O_RDONLY  (0x0000)  /* open for reading */
O_WRONLY  (0x0001)  /* open for writing */
O_RDWR    (0x0002)  /* open for read & write */
O_APPEND  (0x0008)  /* append on each write */
O_CREAT   (0x200)   /* open with file create */
O_TRUNC   (0x400)   /* open with truncation */
O_BINARY  (0x8000)  /* open in binary mode */
```

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

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

  The next available file_descriptor (in order from 3 to 20) is assigned to each new device opened. You can use the finddevice( ) function to return the device structure and use this pointer to search the _stream array for the same pointer. The file_descriptor number is the other member of the _stream array.

Return Value

The function returns one of the following values:

≠−1	if successful
−1	if fails

read — *Read Characters From Buffer*

Syntax for C

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

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

Syntax for C++

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

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

Description

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

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

Return Value

The function returns one of the following values:

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

rename — *Rename File*

Syntax for C

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

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

Syntax for C++

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

int 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

7.2.2 Adding A Device for C I/O

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

1) Define the device-level functions as described in section 7.2.1, *Overview of Low-Level I/O Implementation*, on page 7-4.

> **Note: Use Unique Function Names**
>
> The function names open( ), close( ), read( ), etc. have been used by the low-level routines. Use other names for the device-level functions that you write.

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

name	String for device name
flags	Specifies whether the device supports multiple streams or not
function pointers	Pointers to the device-level functions:

- CLOSE
- LSEEK
- OPEN
- READ
- RENAME
- WRITE
- UNLINK

The first entry in the device table is predefined to be the host device on which the debugger is running. The low-level routine add_device( ) finds the first empty position in the device table and initializes the device fields with the passed in arguments. For a complete description of the add_device function, see page 7-6.

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(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 long my_lseek(int fno, long offset, int origin);
extern int  my_unlink(const char *path);
extern int  my_rename(const char *old_name, char *new_name);

main()
{
   FILE *fid;

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

                                 my_unlink, my_rename);

   fid = fopen(”mydevice:test”,”w”);

   fprintf(fid,”Hello, world\n”);

   fclose(fid);
}
```

7.3 Header Files

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

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

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

assert.h	inttypes.h	setjmp.h	stdio.h
ctype.h	iso646.h	stdarg.h	stdlib.h
errno.h	limits.h	stddef.h	string.h
float.h	math.h	stdint.h	time.h

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

cassert	climits	cstdio	new
cctype	cmath	cstdlib	stdexcept
cerrno	csetjmp	cstring	typeinfo
cfloat	cstdarg	ctime	
ciso646	cstddef	exception	

Furthermore, the following header files are included for the additional functions we provide:

cpy_tbl.h	file.h

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

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

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

Sections 7.3.1 through 7.3.9 describe the header files that are included with the C/C++ compiler. Section 7.4, *Summary of Run-Time-Support Functions and Macros*, on page 7-28, lists the functions that these headers declare.

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

The assert macro is defined as follows:

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

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

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

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). Character-typing functions have names in the form is*xxx* (for example, *isdigit*).

The character conversion functions convert characters to lowercase, uppercase, or ASCII, and return the converted character. Character-typing 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 is passed that has side effects. The typing macros expand to a lookup operation in an array of flags (this array is defined in ctype.c). The macros have the same name as the corresponding functions, but each macro is prefixed with an underscore (for example, *_isdigit*).

See Table 7−3 (b) on page 7-29 for a list of these character-typing and conversion functions.

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)

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.

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

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

How to implement I/O for the MSP430 is described in section 7.2, *The C/IO Functions*, on page 7-3.

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 MSP430'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/climits)

Macro	Value	Description
CHAR_BIT	8	Number of bits in type char
SCHAR_MIN	−128	Minimum value for a signed char
SCHAR_MAX	127	Maximum value for a signed char
UCHAR_MAX	255	Maximum value for an unsigned char
CHAR_MIN	0	Minimum value for a char
CHAR_MAX	UCHAR_MAX	Maximum value for a char
SHRT_MIN	(_SHxT_MAX −1)	Minimum value for a short int
SHRT_MAX	32 767	Maximum value for a short int
USHRT_MAX	65 535	Maximum value for an unsigned short int
INT_MIN	(−INT_MAX − 1)	Minimum value for an int
INT_MAX	32767	Maximum value for an int
UINT_MAX	65535	Maximum value for an unsigned int
LONG_MIN	(−LONG_MAX − 1)	Minimum value for a long int
LONG_MAX	2 147 483 647	Maximum value for a long int
ULONG_MAX	4 294 967 295	Maximum value for an unsigned long int

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

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

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 1028	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	1.192092896e−07f 1.192092896e−07f 1.192092896e−07f	Minimum positive float, double, or long double number *x* such that $1.0 + x \neq 1.0$
FLT_MIN DBL_MIN LDBL_MIN	1.175494351e−38f 1.175494351e−38f 1.175494351e−38f	Minimum positive float, double, or long double
FLT_MAX DBL_MAX LDBL_MAX	3.402823466e+38f 3.402823466e+38f 3.402823466e+38f	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/cfloat header file supplied with the compiler for the full precision carried by the processor.

7.3.6 Format Conversion of Integer Types (inttypes.h)

The stdint.h header declares sets of integer types of specified widths and defines corresponding sets of macros. The inttypes.h header contains stdint.h and also provides a set of integer types with definitions that are consistent across machines and independent of operating systems and other implementation idiosyncrasies. The inttypes.h header declares functions for manipulating greatest-width integers and converting numeric character strings to greatest-width integers.

Through typedef, inttypes.h defines integer types of various sizes. You are free to typedef integer types as standard C integer types or as the types provided in inttypes.h. Consistent use of the inttypes.h header greatly increases the portability of your program across platforms.

The header declares three types:

- The *imaxdiv_t* type is a structure type that is the type of the value returned by the imaxdiv function
- The *intmax_t* type is an integer type large enough to represent any value of any signed integer type
- The *uintmax_t* type is an integer type large enough to represent any value of any unsigned integer type

The header declares several macros and functions:

- For each size type available on the architecture and provided in stdint.h, there are several fprintf and fscanf macros. For example, three fprintf macros for signed integers are PRId32, PRIdLEAST32, and PRIdFAST32. An example use of these macros is:

```
printf(”The largest integer value is %020”
      PRIxMAX “\n”, i);
```

- The *imaxabs* function that computes the absolute value of an integer of type intmax_t.
- The *strtoimax* and *strtoumax* functions, which are equivalent to the strtol, strtoll, strtoul, and strtoull functions. The initial portion of the string is converted to intmax_t and uintmax_t, respectively.

For detailed information on the inttypes.h header, see the *ISO/IEC 9899:1999, International Standard − Programming Languages − C (The C Standard*.

7.3.7 Alternative Spellings (iso646.h/ciso646)

The iso646.h/ciso646 header defines the following eleven macros that expand to the corresponding tokens:

Macro	Token	Macro	Token
and	&&	not_eq	!=
and_eq	&=	or	||
bitand	&	or_eq	|=
bitor	|	xor	^
compl	~	xor_eq	^=
not	!		

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

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

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

These functions are listed in Table 7−3(c) on page 7-30.

7.3.9 Nonlocal Jumps (setjmp.h/csetjmp)

The setjmp.h/csetjmp header defines a type, and a macro, and declares a function for bypassing the normal function call and return discipline. These are listed in Table 7−3(d) on page 7-31 and include:

- 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 its jmp_buf argument for later use by the longjmp function.
- The *longjmp* function uses its jmp_buf argument to restore the program environment.

7.3.10 Variable Arguments (stdarg.h/cstdarg)

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

- The three macros are *va_start*, *va_arg*, and *va_end*. These macros are used when the number and type of arguments can vary each time a function is called.
- The type, *va_list*, is a pointer type that can hold information for va_start, va_end, and va_arg.

A variable-argument function can use the macros declared by stdarg.h/cstdarg to step through its argument list at run time when the function 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) on page 7-31.

7.3.11 Standard Definitions (stddef.h/cstddef)

The stddef.h/cstddef header defines these types and macros:

- The *ptrdiff_t* type is a signed integer type that is the data type resulting from the subtraction of two pointers.
- The *size_t* type is an unsigned integer type that is the data type of the *sizeof* operator.
- The *NULL* macro expands to a null pointer constant (0).
- The *offsetof(type, identifier)* macro expands to an integer that has type *size_t*. The result is the value of an offset in bytes to a structure member (identifier) from the beginning of its structure (type).

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

7.3.12 Integer Types (stdint.h)

The stdint.h header declares sets of integer types of specified widths and defines corresponding sets of macros. It also defines macros that specify limits of integer types that correspond to types defined in other standard headers. Types are defined in these categories:

- Integer types with certain exact widths of the signed form int*N*_t and of the unsigned form uint*N*_t
- Integer types with at least certain specified widths of the signed form int_least*N*_t and of the unsigned form uint_least*N*_t
- Fastest integer types with at least certain specified widths of the signed form int_fast*N*_t and of the unsigned form uint_fast*N*_t
- Signed, *intprt_t*, and unsigned, *uintptr_t*, integer types large enough to hold a pointer value
- Signed, *intmax_t*, and unsigned, *uintmax_t*, integer types large enough to represent any value of any integer type

For each signed type provided by stdint.h there is a macro that specifies the minimum or maximum limit. Each macro name corresponds to a similar type name described above.

The INT*N*_C(*value*) macro expands to a signed integer constant with the specified value and type int_least*N*_t. The unsigned UINT*N*_C(*value*) macro expands to an unsigned integer constant with the specified value and type uint_least*N*_t.

This example shows a macro defined in stdint.h that uses the smallest integer that can hold at least 16 bits:

```
       typedef    uint_least_16 id_number;
extern id_number  lookup_user(char *uname);
```

For detailed information on the stdint.h header, see the *ISO/IEC 9899:1999, International Standard − Programming Languages − C (The C Standard*.

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

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

- The *size_t* type is an unsigned integer type that is the data type of the *sizeof* operator. Originally defined in sttdef.h/cstddef.
- The *fpos_tl* type is a signed integer type that can uniquely specify every position within a file.
- The *FILE* type is a structure type that records all the information necessary to control a stream.

The macros are:

- The *NULL* macro expands to a null pointer constant(0). Originally defined in sttdef.h/cstddef. It is not redefined if it was already defined.
- The *BUFSIZ* macro expands to the size of the buffer that setbuf( ) uses.
- The *EOF*macro is the end-of-file marker.
- The *FOPEN_MAX* macro expands to the largest number of files that can be open at one time.
- The *FILENAME_MAX* macro expands to the length of the longest file name in characters.
- The *L_tmpnam* macro expands to the longest filename string that tmpnam( ) can generate.
- The *SEEK_CUR*, *SEEK_SET*, and *SEEK_END* macros expand to indicate the position (current, start-of-file, or end-of-file, respectively) in a file.
- The *TMP_MAX* macro expands to the maximum number of unique filenames that tmpnam( ) can generate.
- The *stderr*, *stdin*, *stdout* macros are pointers to the standard error, input, and output files, respectively.

The input/output functions are listed in Table 7−3(f) on page 7-31.

7.3.14 General Utilities (stdlib.h/cstdlib)

The stdlib.h/cstdlib header defines a macro and types and declares functions. The macro is named RAND_MAX, and it returns the largest value returned by the rand( ) function. The types are:

- The *div_t* type is a structure type that is the type of the value returned by the div function.
- The *ldiv_t* type is a structure type that is the type of the value returned by the ldiv function

The functions are:

- Memory management functions allow you to allocate and deallocate packets of memory. By default, these functions can use 2K bytes of memory. You can change this amount at link time by invoking the link step with the −−heap_size option and specifying the desired heap size as a constant directly after the option.
- String conversion functions convert strings to numeric representations.
- Searching and sorting functions allow you to search and sort arrays.
- Sequence-generation functions allow you to generate a pseudorandom sequence and allow you to choose a starting point for a sequence.
- Program-exit functions 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 7-34.

7.3.15 String Functions (string.h/cstring)

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

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

In C/C++, all character strings are terminated with a 0 (null) character. The string functions named str*xxx* all operate according to this convention. Additional functions that are also declared in string.h/cstring allow you to perform corresponding operations on arbitrary sequences of bytes (data objects) where a 0 value does not terminate the object. These functions are named mem*xxx*.

When you use functions that move or copy strings, be sure that the destination is large enough to contain the result. The functions are listed in Table 7−3(h) on page 7-35.

7.3.16 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 (0 and up)     */
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 are listed in Table 7−3(i) on page 7-37.

You can adjust local time 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 run-time or by editing tmzone.c and changing the initialization. The default time zone is CST (Central Standard Time), U.S.A.

The basis for all the time.h/ctime functions are the system functions of clock and time. Time provides the current time (in time_t format), and clock provides the system time (in arbitrary units). You can divide the value returned by clock by the macro CLOCKS_PER_SEC to convert it to seconds. Since these functions and the CLOCKS_PER_SEC macro are system specific, only stubs are provided in the library. To use the other time functions, you must supply custom versions of these functions.

Note: Writing Your Own Clock Function

The clock function is host-system specific, so you must write your own clock function. 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.

7.3.17 Exception Handling (exception and stdexcept)

The exception and stdexcept include files are for C++ exception handling.

7.3.18 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.

7.3.19 Run-Time 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 Run-Time-Support Functions and Macros

Table 7−3 summarizes the run-time-support header files (in alphabetical order) provided with the MSP430 ANSI/ISO C/C++ compiler. Most of the functions described are per the ISO standard and behave exactly as described in the standard.

The functions and macros listed in Table 7−3 are described in detail in section 7.5, *Description of Run-Time-Support Functions and Macros*, on page 7-38. For a complete description of a function or macro, see the indicated page.

A superscripted number is used in the following descriptions to show exponents. For example, x^y is the equivalent of x to the power y.

Table 7−3. Summary of Run-Time-Support Functions and Macros

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

Macro	Description	Page
void **assert**(int expression);	Inserts diagnostic messages into programs	7-41

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

Function	Description	Page
int **isalnum**(int c);	Tests c to see if it is an alphanumeric ASCII character	7-61
int **isalpha**(int c);	Tests c to see if it is an alphabetic ASCII character	7-61
int **isascii**(int c);	Tests c to see if it is an ASCII character	7-61
int **iscntrl**(int c);	Tests c to see if it is a control character	7-61
int **isdigit**(int c);	Tests c to see if it is a numeric character	7-61
int **isgraph**(int c);	Tests c to see if it is any printing character except a space	7-61
int **islower**(int c);	Tests c to see if it is a lowercase alphabetic ASCII character	7-61
int **isprint**(int c);	Tests c to see if it is a printable ASCII character (including spaces)	7-61
int **ispunct**(int c);	Tests c to see if it is an ASCII punctuation character	7-61
int **isspace**(int c);	Tests c to see if it is an ASCII spacebar, tab (horizontal or vertical), carriage return, formfeed, or newline character	7-61
int **isupper**(int c);	Tests c to see if it is an uppercase alphabetic ASCII character	7-61
int **isxdigit**(int c);	Tests c to see if it is a hexadecimal digit	7-61
int **toascii**(int c);	Masks c into a legal ASCII value	7-95
int **tolower**(int c);	Converts c to lowercase if it is uppercase	7-95
int **toupper**(int c);	Converts c to uppercase if it is lowercase	7-95

Note: Functions in ctype.h/cctype are expanded inline except when the −−no_inlining option is used.

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

Function	Description	Page
double **acos**(double x);	Returns the arc cosine of x	7-39
double **asin**(double x);	Returns the arc sine of x	7-40
double **atan**(double x);	Returns the arc tangent of x	7-42
double **atan2**(double y, double x);	Returns the inverse tangent of y/x	7-42
double **ceil**(double x);	Returns the smallest integer greater than or equal to x; expands inline except when the −−no_inlining option is used	7-45
double **cos**(double x);	Returns the cosine of x	7-47
double **cosh**(double x);	Returns the hyperbolic cosine of x	7-47
double **exp**(double x);	Returns the exponential function of x; expands inline except when the −−no_inlining option is used	7-50
double **fabs**(double x);	Returns the absolute value of x	7-51
double **floor**(double x);	Returns the largest integer less than or equal to x; expands inline except when the −−no_inlining option is used	7-53
double **fmod**(double x, double y);	Returns the floating-point remainder of x/y; expands inline except when the −−no_inlining option is used	7-54
double **frexp**(double value, int *exp);	Breaks value into a normalized fraction and an integer power of 2	7-57
double **ldexp**(double x, int exp);	Multiplies x by an integer power of 2	7-62
double **log**(double x);	Returns the natural logarithm of x	7-63
double **log10**(double x);	Returns the base-10 (common) logarithm of x	7-64
double **modf**(double value, double *iptr);	Breaks value into a signed integer and a signed fraction	7-70
double **pow**(double x, double y);	Returns x raised to the power y	7-71
double **sin**(double x);	Returns the sine of x	7-78
double **sinh**(double x);	Returns the hyperbolic sine of x	7-78
double **sqrt**(double x);	Returns the nonnegative square root of x	7-79
double **tan**(double x);	Returns the tangent of x	7-93
double **tanh**(double x);	Returns the hyperbolic tangent of x	7-93

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

Macro or Function	Description	Page
int **setjmp**(jmp_buf env);	Saves calling environment for later use by longjmp function; this is a macro	7-76
void **longjmp**(jmp_buf env, int _val);	Uses jmp_buf argument to restore a previously saved program environment	7-76

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

Macro	Description	Page
type **va_arg**(_ap, type);	Accesses the next argument of type *type* in a variable-argument list	7-96
void **va_end**(_ap);	Resets the calling mechanism after using va_arg	7-96
void **va_start**(_ap, parmN);	Initializes ap to point to the first operand in the variable-argument list	7-96

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

Function	Description	Page
void **clearerr**(FILE *_fp);	Clears the EOF and error indicators for the stream that _fp points to	7-46
int **fclose**(FILE *_fp);	Flushes the stream that _fp points to and closes the file associated with that stream	7-51
int **feof**(FILE *_fp);	Tests the EOF indicator for the stream that _fp points to	7-51
int **ferror**(FILE *_fp);	Tests the error indicator for the stream that _fp points to	7-52
int **fflush**(register FILE *_fp);	Flushes the I/O buffer for the stream that _fp points to	7-52
int **fgetc**(register FILE *_fp);	Reads the next character in the stream that _fp points to	7-52
int **fgetpos**(FILE *_fp, fpos_t *_pos);	Stores the object that _pos points to as the current value of the file position indicator for the stream that _fp points to	7-52
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	7-53
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	7-54
int **fprintf**(FILE *_fp, const char *_format, ...);	Writes to the stream that _fp points to	7-54

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

Function	Description	Page
int **fputc**(int _c, register FILE *_fp);	Writes a single character, _c, to the stream that _fp points to	7-55
int **fputs**(const char *_ptr, register FILE *_fp);	Writes the string pointed to by _ptr to the stream pointed to by _fp	7-55
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	7-55
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	7-56
int **fscanf**(FILE *_fp, const char *_fmt, ...);	Reads formatted input from the stream that _fp points to	7-57
int **fseek**(register FILE *_fp, long _offset, int _ptrname);	Sets the file position indicator for the stream that _fp points to	7-57
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.	7-58
long **ftell**(FILE *_fp);	Obtains the current value of the file position indicator for the stream that _fp points to	7-58
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	7-58
int **getc**(FILE *_p);	Reads the next character in the stream that _fp points to	7-59
int **getchar**(void);	A macro that calls fgetc( ) and supplies stdin as the argument	7-59
char ***gets**(char *_ptr);	Performs the same function as fgets( ) using stdin as the input stream	7-60
void **perror**(const char *_s);	Maps the error number in _s to a string and prints the error message	7-70
int **printf**(const char *_format, ...);	Performs the same function as fprintf( ) but uses stdout as its output stream	7-71
int **putc**(int _x, FILE *_fp);	A macro that performs like fputc( )	7-71
int **putchar**(int _x);	A macro that calls fputc( ) and uses stdout as the output stream	7-72
int **puts**(const char *_ptr);	Writes the string pointed to by _ptr to stdout	7-72

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

Function	Description	Page
int **remove** (const char *_file);	Causes the file with the name pointed to by _file to be no longer available by that name	7-74
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	7-75
void **rewind** (register FILE *_fp);	Sets the file position indicator for the stream pointed to by _fp to the beginning of the file	7-75
int **scanf** (const char *_fmt, ...);	Performs the same function as fscanf( ) but reads input from stdin	7-75
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	7-76
int **setvbuf** (register FILE *_fp, register char *_buf, register int _type, register size_t _size);	Defines and associates a buffer with a stream	7-77
int **sprintf** (char *_string, const char *_format, ...);	Performs the same function as fprintf( ) but writes to the array that _string points to	7-79
int **sscanf** (const char *_str, const char *_fmt, ...);	Performs the same function as fscanf( ) but reads from the string that _str points to	7-80
FILE ***tmpfile**(void);	Creates a temporary file	7-94
char ***tmpnam**(char *_s);	Generates a string that is a valid filename (that is, the filename is not already being used)	7-95
int **ungetc** (int _c, register FILE *_fp);	Pushes the character specified by _c back into the input stream pointed to by _fp	7-96
int **vfprintf** (FILE *_fp, const char *_format, va_list _ap);	Performs the same function as fprintf( ) but replaces the argument list with _ap	7-97
int **vprintf** const char *_format, va_list _ap);	Performs the same function as printf( ) but replaces the argument list with _ap	7-97
int **vsprintf** (char *_string, const char *_format, va_list _ap);	Performs the same function as sprintf( ) but replaces the argument list with _ap	7-98

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

Function	Description	Page
void **abort**(void)	Terminates a program abnormally	7-38
int **abs**(int j);	Returns the absolute value of j; expands inline unless −x0 is used	7-39
void **atexit**(void (*fun)(void));	Registers the function pointed to by fun, called without arguments at normal program termination	7-43
double **atof**(const char *st);	Converts a string to a floating-point value; expands inline if −x is used	7-43
int **atoi**(register const char *st);	Converts a string to an integer value	7-43
long **atol**(register const char *st);	Converts a string to a long integer value; expands inline if −x is used	7-43
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	7-44
void ***calloc**(size_t num, size_t size);	Allocates and clears memory for num objects, each of size bytes	7-45
div_t **div**(register int numer, register int denom);	Divides numer by denom producing a quotient and a remainder	7-49
void **exit**(int status);	Terminates a program normally	7-50
void **free**(void *packet);	Deallocates memory space allocated by malloc, calloc, or realloc	7-56
char ***getenv**(const char *_string)	Returns the environment information for the variable associated with _string	7-59
long **labs**(long i);	Returns the absolute value of i; expands inline unless −x0 is used	7-39
long long **llabs**(long long i);	Returns the absolute value of i; expands inline	7-39
ldiv_t **ldiv**(long numer, long denom);	Divides numer by denom	7-62
int **ltoa**(long val, char *buffer);	Converts val to the equivalent string	7-64
void ***malloc**(size_t size);	Allocates memory for an object of size bytes	7-65
void ***memalign**(size_t alignment, size_t size);	Allocates memory for an object of size bytes aligned to an alignment byte boundary	7-65
void **minit**(void);	Resets all the memory previously allocated by malloc, calloc, or realloc	7-68

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

Function	Description	Page
void **qsort**(void *_base, size_t nmemb, size_t size, int (*compar) (void));	Sorts an array of nmemb members; *base* points to the first member of the unsorted array, and *size* specifies the size of each member	7-73
int **rand**(void);	Returns a sequence of pseudorandom integers in the range 0 to RAND_MAX	7-73
void ***realloc**(void *packet, size_t size);	Changes the size of an allocated memory space	7-74
void **srand**(unsigned int seed);	Resets the random number generator	7-73
double **strtod**(const char *st, char **endptr);	Converts a string to a floating-point value	7-91
long **strtol**(const char *st, char **endptr, int base);	Converts a string to a long integer	7-91
long long **strtoll**(const char *st, char **endptr, int base);	Converts a string to a long long integer	7-91
unsigned long **strtoul**(const char *st, char **endptr**,** int base)**;**	Converts a string to an unsigned long integer	7-91
unsigned long long **strtoull**(const char *st, char **endptr**,** int base)**;**	Converts a string to an unsigned long long integer	7-91

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

Function	Description	Page
void ***memchr**(const void *cs, int c, size_t n);	Finds the first occurrence of c in the first n characters of s; expands inline if −x is used	7-66
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	7-66
void ***memcpy**(void *s1, const void *s2, register size_t n);	Copies n characters from s2 to s1	7-67
void ***memmove**(void *s1, const void *s2, size_t n);	Moves n characters from s2 to s1	7-67
void ***memset**(void *mem, register int ch, register size_t length);	Copies the value of ch into the first length characters of mem; expands inline of −x is used	7-67
char ***strcat**(char *string1, const char *string2);	Appends string2 to the end of string1	7-80
char ***strchr**(const char *string, int c);	Finds the first occurrence of character c in s; expands inline if −x is used	7-80
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.	7-81

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

Function	Description	Page
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	7-81
char ***strcpy**(register char *dest, register const char *src);	Copies string src into dest; expands inline if −x is used	7-82
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	7-83
char ***strerror**(int errno);	Maps the error number in errno to an error message string	7-83
size_t **strlen**(char *string);	Returns the length of a string	7-85
char ***strncat**(char *dest, const char *src, register size_t n);	Appends up to n characters from src to dest	7-86
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	7-87
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	7-88
char ***strpbrk**(const char *string, const char *chs);	Locates the first occurrence in string of *any* character from chs	7-89
char ***strrchr**(const char *string, int c);	Finds the last occurrence of character c in string; expands inline if −x is used	7-89
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	7-90
char ***strstr**(register const char *string1, const char *string2);	Finds the first occurrence of string2 in string1	7-90
char ***strtok**(char *str1, const char *str2);	Breaks str1 into a series of tokens, each delimited by a character from str2	7-92
size_t **strxfrm**(register char *to, register const char *from, register size_t n);	Transforms n characters from *from*, to *to*	7-93

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

Function	Description	Page
char ***asctime**(const struct tm *timeptr);	Converts a time to a string	7-40
clock_t **clock**(void);	Determines the processor time used	7-46
char ***ctime**(const time_t *timer);	Converts calendar time to local time	7-48
double **difftime**(time_t time1, time_t time0);	Returns the difference between two calendar times	7-48
struct tm ***gmtime**(const time_t *timer);	Converts calendar time to Greenwich Mean Time	7-60
struct tm ***localtime**(const time_t *timer);	Converts calendar time to local time	7-63
time_t **mktime**(register struct tm *tptr);	Converts local time to calendar time	7-69
size_t **strftime**(char *out, size_t maxsize, const char *format, const struct tm *time);	Formats a time into a character string	7-84
time_t **time**(time_t *timer);	Returns the current calendar time	7-94

7.5 Description of Run-Time-Support Functions and Macros

This section describes the run-time-support functions and macros. For each function or macro, the syntax is given in both C and C++. Because the functions and macros originated from the C header files, however, program examples are shown in C code only. The same program in C++ code would differ in that the types and functions declared in the header file are introduced into the std namespace.

abort — *Abort*

Syntax for C

#include <stdlib.h>

void **abort**(void);

Syntax for C++

#include <cstdlib>

void **std::abort**(void);

Defined in

exit.c in rts.src

Description

The abort function terminates the program.

Example

```
void abort(void)
{
   exit(EXIT_FAILURE);
}
```

See the exit function on page 7-50.

abs/labs/llabs — *Absolute Value*

Syntax for C

#include <stdlib.h>

int **abs**(int j);
long **labs**(long i);
long long **llabs**(long long i);

Syntax for C++

#include <cstdlib>

int **std::abs**(int j);
long **std::labs**(long i);
long long **std::llabs**(long long i);

Defined in

abs.c in rts.src

Description

The C/C++ compiler supports three functions that return the absolute value of an integer:

- The abs function returns the absolute value of an integer j.
- The labs function returns the absolute value of a long integer k.
- The llabs function returns the absolute value of a long long i.

Since int and long are functionally equivalent types in MSP430 C/C++, the abs and labs functions are also functionally equivalent.

acos — *Arc Cosine*

Syntax for C

#include <math.h>

double **acos**(double x);

Syntax for C++

#include <cmath>

double **std::acos**(double x);

Defined in

asin.c in rts.src

Description

The acos function returns the arc cosine of a floating-point argument x, which must be in the range [−1,1]. The return value is an angle in the range [0,π] radians.

Example

```
double realval, radians;

return (realval = 1.0;
radians = acos(realval);
return (radians);   /* acos return pi/2 */
```

asctime — *Internal Time to String*

Syntax for C

#include <time.h>

char ***asctime**(const struct tm *timeptr);

Syntax for C++

#include <ctime>

char ***std::asctime**(const struct tm *timeptr);

Defined in

asctime.c in rts.src

Description

The asctime function converts a broken-down time into a string with the following form:

```
Mon Jan 11 11:18:36 1988 \n\0
```

The function returns a pointer to the converted string.

For more information about the functions and types that the time.h/ctime header declares and defines, see section 7.3.16, *Time Functions (time.h/ctime)*, on page 7-26.

asin — *Arc Sine*

Syntax for C

#include <math.h>

double **asin**(double x);

Syntax for C++

#include <cmath>

double **std::asin**(double x);

Defined in

asin.c in rts.src

Description

The asin function returns the arc sine of a floating-point argument x, which must be in the range [−1, 1]. The return value is an angle in the range [−π/2, π/2] radians.

Example

```
double realval, radians;
realval = 1.0;
radians = asin(realval);  /* asin returns pi/2 */
```

assert — Insert Diagnostic Information

Syntax for C

#include <assert.h>

void **assert**(int expr);

Syntax for C++

#include <cassert>

void **assert**(int expr);

Defined in

assert.h/cassert as macro

Description

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

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

The header file that declares the assert macro refers to another macro, NDEBUG. If you have defined NDEBUG as a macro name when the assert.h header is included in the source file, the assert macro is defined as:

```
#define assert(ignore)
```

The header file that defines the assert macro refers to another macro, NASSERT. If you have defined NASSERT as a macro name when the assert.h header is included in the source file, the assert macro behaves as if it is a call to the _nassert intrinsic.

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(rvalr, rvalu);   /* return value = 0 */
```

atexit — *Register Function Called by Exit ( )*

Syntax for C

#include <stdlib.h>

void **atexit**(void (*fun)(void));

Syntax for C++

#include <cstdlib>

void **std::atexit**(void (*fun)(void));

Defined in

exit.c in rts.src

Description

The atexit function registers the function that is pointed to by *fun*, to be called without arguments at normal program termination. Up to 32 functions can be registered.

When the program exits through a call to the exit function, the functions that were registered are called, without arguments, in reverse order of their registration.

atof/atoi/atol — *Convert 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. In the atof stream, the fractional part of the number follows, then the exponent, including an optional *sign*.

The first character that cannot be part of the number terminates the string.

The functions do not handle any overflow resulting from the conversion.

Because int and long are functionally equivalent in MSP430 C/C++, the atoi and atol functions are also functionally equivalent.

bsearch — *Array Search*

Syntax for C

#include <stdlib.h>

void ***bsearch**(const void *key, const void *base, size_t nmemb, size_t size**,**
int (*compar)(const void *, const void *));

Syntax for C++

#include <cstdlib>

void ***std::bsearch**(const void *key, const void *base, size_t nmemb,
size_t size**,** int (*compar)(const void *, const void *));

Defined in

bsearch.c in rts.src

Description

The bsearch function searches through an array of nmemb objects for a member that matches the object that key points to. Argument base points to the first member in the array; size specifies the size (in bytes) of each member.

The contents of the array must be in ascending order. If a match is found, the function returns a pointer to the matching member of the array; if no match is found, the function returns a null pointer (0).

Argument compar points to a function that compares key to the array elements. The comparison function should be declared as:

```
int  cmp(const void *ptr1, const void *ptr2)
```

The cmp function compares the objects that prt1 and ptr2 point to and returns one of the following values:

< 0	if *ptr1 is less than *ptr2
0	if *ptr1 is equal to *ptr2
> 0	if *ptr1 is greater than *ptr2

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 2K bytes. You can change this amount at link time by invoking the link step with the −−heap_size option and specifying the desired size of the heap (in bytes) directly after the option. See section 6.1.3, *Dynamic Memory Allocation*, on page 6-5.

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.c 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>
```

clock_t std::clock(void);

Defined in clock.c in rts.src

Description The clock function determines the amount of processor time used. It returns an approximation of the processor time used by a program since the program began running. The time in seconds is the return value divided by the value of the macro CLOCKS_PER_SEC.

If the processor time is not available or cannot be represented, the clock function returns the value of [(clock_t) −1].

Note: Writing Your Own Clock Function

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

For more information about the functions and types that the time.h/ctime header declares and defines, see section 7.3.16, *Time Functions (time.h/ctime)*, on page 7-26.

cos — *Cosine*

Syntax for C

#include <math.h>

double **cos**(double x);

Syntax for C++

#include <cmath>

double **std::cos**(double x);

Defined in

cos.c in rts.src

Description

The cos function returns the cosine of a floating-point number x. The angle x is expressed in radians. An argument with a large magnitude can produce a result with little or no significance.

Example

```
double radians, cval; /* cos returns cval */
radians = 3.1415927;
cval = cos(radians);  /* return value = −1.0 */
```

cosh — *Hyperbolic Cosine*

Syntax for C

#include <math.h>

double **cosh**(double x);

Syntax for C++

#include <cmath>

double **std::cosh**(double x);

Defined in

cosh.c in rts.src

Description

The cosh function returns the hyperbolic cosine of a floating-point number x. A range error occurs if the magnitude of the argument is too large.

Example

```
double x, y;

x = 0.0;
y = cosh(x); /* return value = 1.0 */
```

ctime — *Calendar Time*

Syntax for C

```
#include <time.h>

char *ctime(const time_t *timer);
```

Syntax for C++

```
#include <ctime>

char *std::ctime(const time_t *timer);
```

Defined in ctime.c in rts.src

Description The ctime function converts a calendar time (pointed to by timer) to local time in the form of a string. This is equivalent to:

```
asctime(localtime(timer))
```

The function returns the pointer returned by the asctime function.

For more information about the functions and types that the time.h/ctime header declares and defines, see section 7.3.16, *Time Functions (time.h/ctime)*, on page 7-26.

difftime — *Time Difference*

Syntax for C

```
#include <time.h>

double difftime(time_t time1, time_t time0);
```

Syntax for C++

```
#include <ctime>

double std::difftime(time_t time1, time_t time0);
```

Defined in difftime.c in rts.src

Description The difftime function calculates the difference between two calendar times, time1 minus time0. The return value expresses seconds.

For more information about the functions and types that the time.h/ctime header declares and defines, see section 7.3.16, *Time Functions (time.h/ctime)*, on page 7-26.

div/ldiv — *Division*

Syntax for C

#include <stdlib.h>

div_t div(int numer, denom);
ldiv_t ldiv(long numer, denom);

Syntax for C++

#include <cstdlib>

div_t std::div(int numer, denom);
ldiv_t std::ldiv(long numer, denom);

Defined in

div.c in rts.src

Description

These functions support integer division by returning numer (numerator) divided by denom (denominator). You can use these functions to get both the quotient and the remainder in a single operation.

- The div function performs integer division. The input arguments are integers; the function returns the quotient and the remainder in a structure of type div_t. The structure is defined as follows:

```
typedef struct
{
   int  quot;       /*  quotient */
   int  rem;           /* remainder */
} div_t;
```

- The ldiv function performs long integer division. The input arguments are long integers; the function returns the quotient and the remainder in a structure of type ldiv_t. The structure is defined as follows:

```
typedef struct
{
   long int  quot;     /*  quotient */
   long int  rem;      /* remainder */
} ldiv_t;
```

The sign of the quotient is negative if either but not both of the operands is negative. The sign of the remainder is the same as the sign of the dividend.

Because ints and longs are equivalent types in MSP430 C/C++, ldiv and div are also equivalent.

exit — *Normal Termination*

Syntax for C

#include <stdlib.h>

void **exit**(int status);

Syntax for C++

#include <cstdlib>

void **std::exit**(int status);

Defined in

exit.c in rts.src

Description

The exit function terminates a program normally. All functions registered by the atexit function are called in reverse order of their registration. The exit function can accept EXIT_FAILURE as a value. (See the abort function on page 7-38.)

You can modify the exit function to perform application-specific shut-down tasks. The unmodified function simply runs in an infinite loop until the system is reset.

The exit function cannot return to its caller.

exp — *Exponential*

Syntax for C

#include <math.h>

double **exp**(double x);

Syntax for C++

#include <cmath>

double **std::exp**(double x);

Defined in

exp.c in rts.src

Description

The exp function returns the exponential function of real number x. The return value is the number e raised to the power x. A range error occurs if the magnitude of x is too large.

Example

```
double x, y;

x = 2.0;
y = exp(x);         /* y = 7.38, which is e**2.0 */
```

fabs — *Absolute Value*

Syntax for C

#include <math.h>

double **fabs**(double x);

Syntax for C++

#include <cmath>

double **std::fabs**(double x);

Defined in

fabs.c in rts.src

Description

The fabs function returns the absolute value of a floating-point number, x.

Example

```
double x, y;

x = −57.5;
y = fabs(x);        /* return value = +57.5 */
```

fclose — *Close File*

Syntax for C

#include <stdio.h>

int **fclose**(FILE *_fp);

Syntax for C++

#include <cstdio>

int **std::fclose**(FILE *_fp);

Defined in

fclose.c in rts.src

Description

The fclose function flushes the stream that _fp points to and closes the file associated with that stream.

feof — *Test EOF Indicator*

Syntax for C

#include <stdio.h>

int **feof**(FILE *_fp);

Syntax for C++

#include <cstdio>

int **std::feof**(FILE *_fp);

Defined in

feof.c in rts.src

Description

The feof function tests the EOF indicator for the stream pointed to by _fp.

ferror — *Test Error Indicator*

Syntax for C

```
#include <stdio.h>
int ferror(FILE *_fp);
```

Syntax for C++

```
#include <cstdio>
int std::ferror(FILE *_fp);
```

Defined in ferror.c in rts.src

Description The ferror function tests the error indicator for the stream pointed to by _fp.

fflush — *Flush I/O Buffer*

Syntax for C

```
#include <stdio.h>
int fflush(register FILE *_fp);
```

Syntax for C++

```
#include <cstdio>
int std::fflush(register FILE *_fp);
```

Defined in fflush.c in rts.src

Description The fflush function flushes the I/O buffer for the stream pointed to by _fp.

fgetc — *Read Next Character*

Syntax for C

```
#include <stdio.h>
int fgetc(register FILE *_fp);
```

Syntax for C++

```
#include <cstdio>
int std::fgetc(register FILE *_fp);
```

Defined in fgetc.c in rts.src

Description The fgetc function reads the next character in the stream pointed to by _fp.

fgetpos — *Store Object*

Syntax for C

```
#include <stdio.h>
int fgetpos(FILE *_fp, fpos_t *pos);
```

Syntax for C++

```
#include <cstdio>
int std::fgetpos(FILE *_fp, fpos_t *pos);
```

Defined in fgetpos.c in rts.src

Description The fgetpos function stores the object pointed to by pos to the current value of the file position indicator for the stream pointed to by _fp.

fgets — *Read Next Characters*

Syntax for C

#include <stdio.h>

char ***fgets**(char *_ptr, register int _size, register FILE *_fp);

Syntax for C++

#include <cstdio>

char ***std::fgets**(char *_ptr, register int _size, register FILE *_fp);

Defined in

fgets.c in rts.src

Description

The fgets function reads the specified number of characters from the stream pointed to by _fp. The characters are placed in the array named by _ptr. The number of characters read is _size −1.

floor — *Floor*

Syntax for C

#include <math.h>

double **floor**(double x);

Syntax for C++

#include <cmath>

double **std::floor**(double x);

Defined in

floor.c in rts.src

Description

The floor function returns a floating-point number that represents the largest integer less than or equal to x.

Example

```
double answer;

answer = floor(3.1415);       /* answer =  3.0 */
answer = floor(-3.5);         /* answer = -4.0 */
```

fmod — *Floating-Point Remainder*

Syntax for C

#include <math.h>

double **fmod**(double x, double y);

Syntax for C++

#include <cmath>

double **std::fmod**(double x, double y);

Defined in

fmod.c in rts.src

Description

The fmod function returns the floating-point remainder of x divided by y. If y = = 0, the function returns 0.

Example

```
double x, y, r;

x = 11.0;
y = 5.0;
r = fmod(x, y);        /* fmod returns 1.0 */
```

fopen — *Open File*

Syntax for C

#include <stdio.h>

FILE ***fopen**(const char *_fname, const char *_mode);

Syntax for C++

#include <cstdio>

FILE ***std::fopen**(const char *_fname, const char *_mode);

Defined in

fopen.c in rts.src

Description

The fopen function opens the file that _fname points to. The string pointed to by _mode describes how to open the file.

fprintf — *Write Stream*

Syntax for C

#include <stdio.h>

int **fprintf**(FILE *_fp, const char *_format, ...);

Syntax for C++

#include <cstdio>

int **std::fprintf**(FILE *_fp, const char *_format, ...);

Defined in

fprintf.c in rts.src

Description

The fprintf function writes to the stream pointed to by _fp. The string pointed to by _format describes how to write the stream.

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 fread(void *_ptr, size_t size, size_t count, FILE *_fp);

Syntax for C++

#include <cstdio>

size std::fread(void *_ptr, size_t size, size_t count, FILE *_fp);

Defined in

fread.c in rts.src

Description

The fread function reads from the stream pointed to by _fp. The input is stored in the array pointed to by _ptr. The number of objects read is _count. The size of the objects is _size.

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 section 6.1.3, *Dynamic Memory Allocation*, on page 6-5.

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 std::ftell(FILE *_fp);
```

Defined in ftell.c in rts.src

Description The ftell function gets the current value of the file position indicator for the stream pointed to by _fp.

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 fgetenv.c in rts.src

Description The getenv function returns the environment information for the variable associated with _string.

> **Note: The getenv Function Is Target-System Specific**
>
> The getenv function is target-system specific, so you must write your own getenv function.

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. Use the function fgets() instead of gets when possible.

gmtime — *Greenwich Mean Time*

Syntax for C

#include <time.h>

struct tm *gmtime(const time_t *timer);

Syntax for C++

#include <ctime>

struct tm *std::gmtime(const time_t *timer);

Defined in

gmtime.c in rts.src

Description

The gmtime function converts a calendar time (pointed to by timer) into a broken-down time, which is expressed as Greenwich Mean Time.

For more information about the functions and types that the time.h/ctime header declares and defines, see section 7.3.16, *Time Functions (time.h/ctime)*, on page 7-26.

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/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/llabs

See abs/labs/llabs on page 7-39.

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^{exp} 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 7-49.

localtime — *Local Time*

Syntax for C

#include <time.h>

struct tm *localtime(const time_t *timer);

Syntax for C++

#include <ctime>

struct tm *std::localtime(const time_t *timer);

Defined in

localtime.c in rts.src

Description

The localtime function converts a calendar time (pointed to by timer) into a broken-down time, which is expressed as local time. The function returns a pointer to the converted time.

For more information about the functions and types that the time.h/ctime header declares and defines, see section 7.3.16, *Time Functions (time.h/ctime)*, on page 7-26.

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 7-76.*

ltoa — *Long Integer to ASCII*

Syntax for C

no prototype provided

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

Syntax for C++

no prototype provided

int **std::ltoa**(long val, char *buffer);

Defined in

ltoa.c in rts.src

Description

The ltoa function is a nonstandard (non-ANSI) function and is provided for compatibility with non-ANSI code. The standard equivalent is sprintf. The function is not prototyped in rts.src. The ltoa function converts a long integer n to an equivalent ASCII string and writes it into the buffer. If the input number val is negative, a leading minus sign is output. The ltoa function returns the number of characters placed in the buffer.

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 2K bytes. You can change this amount at link time by invoking the link step with the −−heap_size option and specifying the desired size of the heap (in bytes) directly after the option. For more information, see section 6.1.3, *Dynamic Memory Allocation*, on page 6-5.

memalign — *Align Heap*

Syntax for C

#include <stdlib.h>

void ***memalign**(size_t _aln, size_t _size);

Syntax for C++

#include <cstdlib>

void ***std::memalign**(size_t _aln, size_t _size);

Defined in

memory.c in rts.src

Description

The memalign function performs like the ANSI/ISO standard malloc function, except that it returns a pointer to a block of memory that is aligned to an *alignment* byte boundary. Thus if _size is 128 and alignment is 16, memalign returns a pointer to a 128-byte block of memory aligned on a 16-byte boundary.

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 ***std::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.

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 2K bytes. You can change this amount at link time by invoking the link step with the −−heap_size option and specifying the desired size of the heap (in bytes) directly after the option. For more information, refer to section 6.1.3, *Dynamic Memory Allocation*, on page 6-5.

Note: No Previously Allocated Objects are Available After minit

Calling the minit function makes *all* the memory space in the heap available again. Any objects that you allocated previously will be lost; do not try to access them.

mktime — *Convert to Calendar Time*

Syntax for C

#include <time.h>

time_t ***mktime**(struct tm *timeptr);

Syntax for C++

#include <ctime>

time_t ***std::mktime**(struct tm *timeptr);

Defined in

mktime.c in rts.src

Description

The mktime function converts a broken-down time, expressed as local time, into proper calendar time. The timeptr argument points to a structure that holds the broken-down time.

The function ignores the original values of tm_wday and tm_yday and does not restrict the other values in the structure. After successful completion of time conversions, tm_wday and tm_yday are set appropriately, and the other components in the structure have values within the restricted ranges. The final value of tm_mday is not sent until tm_mon and tm_year are determined.

The return value is encoded as a value of type time_t. If the calendar time cannot be represented, the function returns the value −1.

For more information about the functions and types that the time.h/ctime header declares and defines, see section 7.3.16, *Time Functions (time.h/ctime)*, on page 7-26.

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 */
```

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 $y \le 0$, 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 <cstdlib>

int **std::putc**(int _x, FILE *_fp);

Defined in

putc.c in rts.src

Description

The putc function writes a character to the stream pointed to by _fp.

putchar — *Write Character to Standard Output*

Syntax for C

```
#include <stdlib.h>

int putchar(int _x);
```

Syntax for C++

```
#include <cstdlib>

int std::putchar(int _x);
```

Defined in putchar.c in rts.src

Description The putchar function writes a character to the standard output device.

puts — *Write to Standard Output*

Syntax for C

```
#include <stdlib.h>

int puts(const char *_ptr);
```

Syntax for C++

```
#include <cstdlib>

int std::puts(const char *_ptr);
```

Defined in puts.c in rts.src

Description The puts function writes the string pointed to by _ptr to the standard output device.

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

These 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 2K bytes. You can change this amount at link time by invoking the link step with the −−−heap_size option and specifying the desired size of the heap (in bytes) directly after the option. For more information, see section 6.1.3, *Dynamic Memory Allocation*, on page 6-5.

remove — *Remove File*

Syntax for C

```
#include <stdlib.h>

int remove(const char *_file);
```

Syntax for C++

```
#include <cstdlib>

int std::remove(const char *_file);
```

Defined in remove.c in rts.src

Description The remove function makes the file pointed to by _file no longer available by that name.

rename — *Rename File*

Syntax for C

```
#include <stdlib.h>

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

Syntax for C++

```
#include <cstdlib>

int std::rename(const char *old_name, const char *new_name);
```

Defined in rename.c in rts.src

Description The rename function renames the file pointed to by old_name. The new name is pointed to by new_name.

rewind — *Position File-Position Indicator to Beginning of File*

Syntax for C

```
#include <stdlib.h>

void rewind(register FILE *_fp);
```

Syntax for C++

```
#include <cstdlib>

void std::rewind(register FILE *_fp);
```

Defined in rewind.c in rts.src

Description The rewind function sets the file position indicator for the stream pointed to by _fp to the beginning of the file.

scanf — *Read Stream From Standard Input*

Syntax for C

```
#include <stdlib.h>

int scanf(const char *_fmt, ...);
```

Syntax for C++

```
#include <cstdlib>

int std::scanf(const char *_fmt, ...);
```

Defined in fscanf.c in rts.src

Description The scanf function reads from the stream from the standard input device. The string pointed to by _fmt describes how to read the stream.

setbuf

Specify Buffer for Stream

Syntax for C

```
#include    <stdlib.h>

void setbuf(register FILE *_fp, char *_buf);
```

Syntax for C++

```
#include    <cstdlib>

void std::setbuf(register FILE *_fp, char *_buf);
```

Defined in setbuf.c in rts.src

Description The setbuf function specifies the buffer used by the stream pointed to by _fp. If _buf is set to null, buffering is turned off. No value is returned.

setjmp/longjmp

Nonlocal Jumps

Syntax for C

```
#include <setjmp.h>

int setjmp(jmp_buff env)
void longjmp(Jn_buf env, int _val)
```

Syntax for C++

```
#include <csetjmp>

int setjmp(jmp_buf env)
void std::longjmp(jmp_buf env, int _val)
```

Defined in setjmp16.asm and setjmp32.asm in rts.src

Description The setjmp.h header defines one type, one macro, and one function for bypassing the normal function call and return discipline:

- The **jmp_buf** type is an array type suitable for holding the information needed to restore a calling environment.
- The **setjmp** macro saves its calling environment in the jmp_buf argument for later use by the longjmp function.

  If the return is from a direct invocation, the setjmp macro returns the value 0. If the return is from a call to the longjmp function, the setjmp macro returns a nonzero value.
- The **longjmp** function restores the environment that was saved in the jmp_buf argument by the most recent invocation of the setjmp macro. If the setjmp macro was not invoked, or if it terminated execution irregularly, the behavior of longjmp is undefined.

  After longjmp is completed, the program execution continues as if the corresponding invocation of setjmp had just returned the value specified by _val. The longjmp function does not cause setjmp to return a value of 0, even if _val is 0. If _val is 0, the setjmp macro returns the value 1.

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 <stdlib.h>

int **setvbuf**(register FILE *_fp, register char *_buf, register int _type, register size_t _size);

Syntax for C++

#include <cstdlib>

int **std::setvbuf**(register FILE *_fp, register char *_buf, register int _type, register size_t _size);

Defined in

setvbuf.c in rts.src

Description

The setvbuf function defines and associates the buffer used by the stream pointed to by _fp. If _buf is set to null, a buffer is allocated. If _buf names a buffer, that buffer is used for the stream. The _size specifies the size of the buffer. The _type specifies the type of buffering as follows:

_IOFBF	Full buffering occurs
_IOLBF	Line buffering occurs
_IONBF	No buffering occurs

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 <stdlib.h>

int **sprintf**(char _string, const char *_format, ...);

Syntax for C++

#include <cstdlib>

int **std::sprintf**(char _string, const char *_format, ...);

Defined in

sprintf.c in rts.src

Description

The sprintf function writes to the array pointed to by _string. The string pointed to by _format describes how to write the stream.

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 7-73.*

sscanf — *Read Stream*

Syntax for C

#include <stdlib.h>

int **sscanf**(const char *str, const char *format, ...);

Syntax for C++

#include <cstdlib>

int **std::sscanf**(const char *str, const char *format, ...);

Defined in

sscanf.c in rts.src

Description

The sscanf function reads from the string pointed to by str. The string pointed to by _format describes how to read the stream.

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/ISO C/C++.

The functions return one of the following values:

< 0	if *string1 is less than *string2
0	if *string1 is equal to *string2
> 0	if *string1 is greater than *string2

Example

```
char *stra = ”why ask why”;
char *strb = ”just do it”;
char *strc = ”why ask why”;

if (strcmp(stra, strb) > 0)
   {
      /*    statements here will be executed    */
   }
if (strcoll(stra, strc) == 0)
   {
      /* statements here will be executed also */
   }
```

strcpy — *String Copy*

Syntax for C

#include <string.h>

char ***strcpy**(register char *dest, register const char *src);

Syntax for C++

#include <cstring>

char ***std::strcpy**(register char *dest, register 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(register const char *string, const char *chs);

Syntax for C++

#include <cstring>

size_t std::strcspn(register 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 C 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

Character	Expands to
%y	The *year* without century as a decimal number (00−99)
%Y	The *year* with century as a decimal number
%Z	The *time zone* name, or by no characters if no time zone exists

For more information about the functions and types that the time.h/ctime header declares and defines, see section 7.3.16, *Time Functions (time.h/ctime)*, on page 7-26.

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/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 string2 with string1. The function returns one of the following values:

< 0	if *string1 is less than *string2
0	if *string1 is equal to *string2
> 0	if *string1 is greater than *string2

Example

```
char *stra = ”why ask why”;
char *strb = ”just do it”;
char *strc = ”why not?”;
size_t size = 4;

if (strncmp(stra, strb, size) > 0)
   {
      /* statements here will get executed */
   }
if (strncmp(stra, strc, size) == 0)
   {
      /* statements here will get executed also */
   }
```

strncpy — *String Copy*

Syntax for C

#include <string.h>

char ***strncpy**(register char *dest, register const char *src**,**
register size_t n);

Syntax for C++

#include <cstring>

char ***std::strncpy**(register char *dest, register const char *src**,**
register size_t n);

Defined in

strncpy.c 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 = strcspn(stra,strb);    /* length = 3 */
length = strcspn(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 does not find the string, it returns a null pointer. If string2 points to a string with length 0, strstr returns string1.

Example

```
char *stra = "so what do you want for nothing?";
char *strb = "what";
char *ptr;

ptr = strstr(stra,strb);
```

The pointer *ptr now points to the w in what in the first string.

strtod/strtol/ strtoll/strtoul/ strtoull

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);
long long **strtoll**(const char *st, char **endptr, int base);
unsigned long **strtoul**(const char *st, char **endptr, int base);
unsigned long long **strtoull**(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);
long long **strtoll**(const char *st, char **endptr, int base);
unsigned long **std::strtoul**(const char *st, char **endptr, int base);
unsigned long long **std::strtoull**(const char *st, char **endptr, int base);

Defined in

strtod.c, strtol.c, strtoll.c, strtoul.c, and strtoull.c in rts.src

Description

Three functions convert ASCII strings to numeric values. For each function, argument st points to the original string. Argument endptr points to a pointer; the functions set this pointer to point to the first character after the converted string.The functions that convert to integers also have a third argument, base, which tells the function the base in which to interpret the string.

- The strtod function converts a string to a floating-point value. The string must have the following format:

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

  The function returns the converted string; if the original string is empty or does not have the correct format, the function returns a 0. If the converted string would cause an overflow, the function returns ±HUGE_VAL; if the converted string would cause an underflow, the function returns 0. If the converted string overflows or underflows, errno is set to the value of ERANGE.

- The strtol function converts a string to a long integer. The string must have the following format:

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

- The strtoll function converts a string to a long 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 strtoull function converts a string to an unsigned long long integer. Specify the string in 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 <cstring>

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 following example, the pointer stra points to the string excuse\0 because strtok has inserted a null character where the first space used to be. In the comments, the notation \0 represents the null character.

```
char *stra = ”excuse me while I kiss the sky”;
char *ptr;

ptr = strtok (stra,” ”);   /* ptr --> ”excuse\0”   */
ptr = strtok (0,” ”);      /* ptr --> ”me\0”       */
ptr = strtok (0,” ”);      /* ptr --> ”while\0”    */
```

strxfrm — *Convert Characters*

Syntax for C

#include <string.h>

size_t strxfrm(char *to, const char *from, size_t n);

Syntax for C++

#include <cstring>

size_t std::strxfrm(char *to, const char *from, size_t n);

Defined in

strxfrm.c in rts.src

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/ctime header declares and defines, see section 7.3.16, *Time Functions (time.h/ctime)*, on page 7-26.

> **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 <stdlib.h>

FILE *tmpfile(void);
```

Syntax for C++

```
#include <cstdlib>

FILE *std::tmpfile(void);
```

Defined in tmpfile.c in rts.src

Description The tmpfile function creates a temporary file.

tmpnam — *Generate Valid Filename*

Syntax for C

```
#include <stdlib.h>

char *tmpnam(char *_s);
```

Syntax for C++

```
#include <cstdlib>

char *std::tmpnam(char *_s);
```

Defined in tmpnam.c in rts.src

Description The tmpnam function generates a string that is a valid filename.

toascii — *Convert to ASCII*

Syntax for C

```
#include <ctype.h>

int toascii(int c);
```

Syntax for C++

```
#include <cctype>

int std::toascii(int c);
```

Defined in toascii.c in rts.src

Description The toascii function ensures that c is a valid ASCII character by masking the lower seven bits. There is also an equivalent macro call _toascii.

tolower/toupper — *Convert Case*

Syntax for C

```
#include <ctype.h>

int tolower(int c);
int toupper(int c);
```

Syntax for C++

```
#include <cctype>

int std::tolower(int c);
int std::toupper(int c);
```

Defined in tolower.c and toupper.c in rts.src

Description Two functions convert the case of a single alphabetic character c into uppercase or lowercase:

- The tolower function converts an uppercase argument to lowercase. If c is already in lowercase, tolower returns it unchanged.
- The toupper function converts a lowercase argument to uppercase. If c is already in uppercase, toupper returns it unchanged.

The functions have macro equivalents named _tolower and _toupper.

ungetc

Write Character to Stream

Syntax for C

#include <stdlib.h>

int **ungetc**(int c, FILE *_fp);

Syntax for C++

#include <cstdlib>

int **std::ungetc**(int c, FILE *_fp);

Defined in

ungetc.c in rts.src

Description

The ungetc function writes the character c to the stream pointed to by _fp.

va_arg/va_end/ va_start

Variable-Argument Macros

Syntax for C

#include <stdarg.h>

typedef char ***va_list;**
type **va_arg**(_ap**,** _type);
void **va_end**(_ap);
void **va_start**(_ap**,** parmN);

Syntax for C++

#include <cstdarg>

typedef char ***va_list;**
type **va_arg**(_ap**,** _type);
void **va_end**(_ap);
void **va_start**(_ap**,** parmN);

Defined in

stdarg.h in rts.src

Description

Some functions are called with a varying number of arguments that have varying types. Such a function, called a variable-argument function, can use the following macros to step through its argument list at run time. The _ap parameter points to an argument in the variable-argument list.

- The va_start macro initializes _ap to point to the first argument in an argument list for the variable-argument function. The parmN parameter points to the rightmost parameter in the fixed, declared list.
- The va_arg macro returns the value of the next argument in a call to a variable-argument function. Each time you call va_arg, it modifies _ap so that successive arguments for the variable-argument function can be returned by successive calls to va_arg (va_arg modifies _ap to point to the next argument in the list). The type parameter is a type name; it is the type of the current argument in the list.
- The va_end macro resets the stack environment after va_start and va_arg are used.

You must call va_start to initialize _ap before calling va_arg or va_end.

Example

```
int    printf (char *fmt...)
       va_list ap;
       va_start(ap, fmt);
       .
       .
       .
       i = va_arg(ap, int);      /* Get next arg, an integer */
       s = va_arg(ap, char *);   /* Get next arg, a string   */
       l = va_arg(ap, long);     /* Get next arg, a long     */
       .
       .
       .
       va_end(ap);               /* Reset                    */
}
```

vfprintf

Write to Stream

Syntax for C

#include <stdlib.h>

int **vfprintf**(FILE *_fp, const char *_format, va list _ap);

Syntax for C++

#include <cstdlib>

int **std::vfprintf**(FILE *_fp, const char *_format, va list _ap);

Defined in

vfprintf.c in rts.src

Description

The vfprintf function writes to the stream pointed to by _fp. The string pointed to by format describes how to write the stream. The argument list is given by _ap.

vprintf

Write to Standard Output

Syntax for C

#include <stdlib.h>

int **vprintf**(const char *_format, va list _ap);

Syntax for C++

#include <cstdlib>

int **std::vprintf**(const char *_format, va list _ap);

Defined in

vprintf.c in rts.src

Description

The vprintf function writes to the standard output device. The string pointed to by _format describes how to write the stream. The argument list is given by _ap.

vsprintf — *Write Stream*

Syntax for C #include <stdlib.h>

int **vsprintf**(char *string, const char *_format, va list _ap)**;**

Syntax for C++ #include <cstdlib>

int **std::vsprintf**(char *string, const char *_format, va list _ap;

Defined in vsprintf.c in rts.src

Description The vsprintf function writes to the array pointed to by _string. The string pointed to by _format describes how to write the stream. The argument list is given by _ap.

Chapter 8

Library-Build Process

When using the MSP430 C/C++ compiler, you can compile your code under a number of different configurations and options that are not necessarily compatible with one another. Because it would be cumbersome to include all possible combinations in individual run-time-support libraries, this package includes a basic run-time-support library, rts430.lib. Also included are library versions that support MSPX devices and versions that support C++ exception handling.

You can also build your own run-time-support libraries using the self-contained run-time-support build process, which is found in rtssrc.zip. This process is described in this chapter, and the archiver is described in the *MSP430 Assembly Language Tools User's Guide.*

Topic **Page**

8.1 Required Non-Texas Instruments Software

To use the self-contained run-time-support build process to rebuild a library with custom options, the following support items are required:

- Perl version 5.6 or later available as perl

  Perl is a high-level programming language designed for process, file, and text manipulation. It is:

  - Generally available from http://www.perl.org/get.htm
  - Available from ActiveState.com as ActivePerl for the PC
  - Available as part of the Cygwin package for the PC

  It must be installed and added to PATH so it is available at the command-line prompt as perl. To ensure perl is available, open a Command Prompt window and execute:

  ```
  perl –v
  ```

  No special or additional Perl modules are required beyond the standard perl module distribution.

- GNU-compatible command-line make tool, such as gmake

  More information is available from GNU at http://www.gnu.org/software/make. This file requires a host C compiler to build. GNU make (gmake) is shipped as part of Code Composer Studio on Windows. GNU make is also included in some Unix support packages for Windows, such as the MKS Toolkit, Cygwin, and Interix. The GNU make used on Windows platforms should explicity report *This program built for Windows32* when the following is executed from the Command Prompt window:

  ```
  gmake –h
  ```

8.2 Using the Library-Build Process

Once the perl and gmake tools are available, unzip the rtssrc.zip into a new, empty directory. See the Makefile for additional information on how to customize a library build by modifying the LIBLIST and/or the OPT_XXX macros.

8.2.1 The Base Option Sets for Building the Libraries

For MSP430 these libraries can be built:

rts430.lib	C run-time support without exception handling
rts430X.lib	MSPX C run-time support without exception handling
rts430_eh.lib	C run-time support with exception handling
rts430X_eh.lib	MSPX C run-time support with exception handling

8.2.2 Rebuild the Desired Library

Once the desired changes have been made, simply use the following syntax from the command-line while in the rtssrc.zip top level directory to rebuild the selected *rtsname* library.

gmake *rtsname*

To use custom options to rebuild a library, simply change the list of options for the appropriate base listed in section 8.2.1 and then rebuild the library. See Table 2−1, *Compiler Options Summary*, beginning on page 2-6, for a summary of available generic and MSP430-specific options.

To build an library with a completely different set of options, define a new OPT_XXX base, choose the type of library from those listed in section 8.2.1, and then rebuild the library. Not all library types are supported by all targets. You may need to make changes to targets_rts_cfg.pm to ensure the proper files are included in your custom library.

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 link step output, it can be difficult to associate a mangled name with its corresponding name in the C++ source code. The C++ *name demangler* is a debugging aid that translates each mangled name it detects to its original name found in the C++ source code.

This chapter tells you how to invoke and use the C++ name demangler. The C++ name demangler reads in input, looking for mangled names. All unmangled text is copied to output unaltered. All mangled names are demangled before being copied to output.

Topic **Page**

9.1 Invoking the C++ Name Demangler

To invoke the C++ name demangler, enter:

dem430 [*options*] [*filenames*]

dem430	Command that runs the name demangler
options	Options affect how the demangler behaves. Options can appear anywhere on the command line or in a link step 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, dem430 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 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*	Outputs to the given *file* rather than to standard out
−u	External names do not have a C++ prefix
−v	Enables verbose mode (output banner)

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 MSP430 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

```
class banana {
public:
   int calories(void);
   banana();
   ~banana();
};

int calories_in_a_banana(void)
{
   banana x;
   return x.calories();
}
```

(b) Resulting assembly for calories_in_a_banana

```
calories_in_a_banana__Fv:
;* ----------------------------------------------------------------------------*
        SUB.W      #4,SP
        MOV.W      SP,r12                 ; |10|
        ADD.W      #2,r12                 ; |10|
        CALL       #__ct__6bananaFv       ; |10|
                                            ; |10|
        MOV.W      SP,r12                 ; |11|
        ADD.W      #2,r12                 ; |11|
        CALL       #calories__6bananaFv   ; |11|
                                            ; |11|
        MOV.W      r12,0(SP)              ; |11|
        MOV.W      SP,r12                 ; |11|
        ADD.W      #2,r12                 ; |11|
        MOV.W      #2,r13                 ; |11|
        CALL       #__dt__6bananaFv       ; |11|
                                            ; |11|
        MOV.W      0(SP),r12              ; |11|
        ADD.W      #4,SP
        RET
```

Executing the C++ name demangler utility demangles all names that it believes to be mangled. If you enter:

% dem430 banana.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

```
calories_in_a_banana():
;* ----------------------------------------------------------------------------*
        SUB.W     #4,SP
        MOV.W     SP,r12                ; |10|
        ADD.W     #2,r12                ; |10|
        CALL      #banana::banana()      ; |10|
                                          ; |10|
        MOV.W     SP,r12                ; |11|
        ADD.W     #2,r12                ; |11|
        CALL      #banana::calories()  ; |11|
                                          ; |11|
        MOV.W     r12,0(SP)             ; |11|
        MOV.W     SP,r12                ; |11|
        ADD.W     #2,r12                ; |11|
        MOV.W     #2,r13                ; |11|
        CALL      #banana::~banana()      ; |11|
                                          ; |11|
        MOV.W     0(SP),r12             ; |11|
        ADD.W     #4,SP
        RET
```

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 link step 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/C++ variables (contained in the .cinit section) before program execution begins.

autoinitialization at load time: An autoinitialization method used by the link step when linking C/C++ code. The link step uses this method when you invoke the link step with the −−ram_model option. This method initializes variables at load time instead of run time.

autoinitialization at run time: An autoinitialization method used by the link step when linking C/C++ code. The link step uses this method when you invoke the link step with the −−rom_model option. The link step loads the .cinit section of data tables into memory, and variables are initialized at run time.

B

BIS: *Bit instruction set*.

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: A sequence of eight adjacent bits operated upon as a unit.

C

C/C++ compiler: A software program that translates C/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.

COFF: See *common object file format*

command file: A file that contains link step or hex conversion utility options and names input files for the link step 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.

compiler program: A utility that lets you compile, assemble, and optionally link in one step. The compiler runs one or more source modules (including the parser, optimizer, and code generator), the assembler, and the link step.

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 run time. This is accomplished by defining a large memory pool (heap) and using the functions to allocate memory from the heap.

E

emulator: A hardware development system that emulates MSP430 operation.

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/C++ preprocessor that is merged with the parser, allowing for faster compilation. Standalone preprocessing or preprocessed listing is also available.

interlist feature: A utility that inserts as comments your original C/C++ source statements into the assembly language output from the assembler. The C/C++ statements are inserted next to the equivalent assembly instructions.

K

K&R C: Kernighan and Ritchie C, the de facto standard as defined in the first edition of *The C Programming Language* (K&R). Most K&R C programs written for earlier, non-ANSI C compilers 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.

link step: 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 link step, 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/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 link step adjusts all the references to a symbol when the symbol's address changes.

run-time 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.

run-time-support functions: Standard ANSI functions that perform tasks that are not part of the C/C++ language (such as memory allocation, string conversion, and string searches).

run-time-support library: A library file, rtsc.src or rtscpp.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.

source file: A file that contains C code, C++ code, or assembly language code that is compiled or assembled to form an object file.

stand-alone preprocessor: A software tool that expands macros, #include files, and conditional compilation as an independent program. It also performs integrated preprocessing, which includes parsing of instructions.

static variable: A 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 3-character sequence that has a meaning (as defined by the ISO 646-1983 Invariant Code Set). These characters cannot be represented in the C character set and are expanded to one character. For example, the 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.

veneer: A sequence of instructions that serves as an alternate entry point into a routine if a state change is required.

Index

A

D

G

J

K

L

M

P

Q

R

W

X