

# ***TMS470R1x Optimizing C/C++ Compiler User's Guide***

Literature Number: SPNU151B  
July 2004



Printed on Recycled Paper

## 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

Amplifiers	<a href="http://amplifier.ti.com">amplifier.ti.com</a>
Data Converters	<a href="http://dataconverter.ti.com">dataconverter.ti.com</a>
DSP	<a href="http://dsp.ti.com">dsp.ti.com</a>
Interface	<a href="http://interface.ti.com">interface.ti.com</a>
Logic	<a href="http://logic.ti.com">logic.ti.com</a>
Power Mgmt	<a href="http://power.ti.com">power.ti.com</a>
Microcontrollers	<a href="http://microcontroller.ti.com">microcontroller.ti.com</a>

### Applications

Audio	<a href="http://www.ti.com/audio">www.ti.com/audio</a>
Automotive	<a href="http://www.ti.com/automotive">www.ti.com/automotive</a>
Broadband	<a href="http://www.ti.com/broadband">www.ti.com/broadband</a>
Digital Control	<a href="http://www.ti.com/digitalcontrol">www.ti.com/digitalcontrol</a>
Military	<a href="http://www.ti.com/military">www.ti.com/military</a>
Optical Networking	<a href="http://www.ti.com/opticalnetwork">www.ti.com/opticalnetwork</a>
Security	<a href="http://www.ti.com/security">www.ti.com/security</a>
Telephony	<a href="http://www.ti.com/telephony">www.ti.com/telephony</a>
Video & Imaging	<a href="http://www.ti.com/video">www.ti.com/video</a>
Wireless	<a href="http://www.ti.com/wireless">www.ti.com/wireless</a>

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

Copyright © 2004, Texas Instruments Incorporated

# Read This First

---

---

---

### ***About This Manual***

The *TMS470R1x Optimizing C/C++ Compiler User's Guide* explains how to use these compiler tools:

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

The TMS470R1x 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 TMS470R1x 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:

- ☐ The TMS470R1x device is referred to as the TMS470.
- ☐ The TMS470 16-bit instruction set is referred to as 16-BIS.
- ☐ The TMS470 32-bit instruction set is referred to as 32-BIS.

- 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:

**cl470** [*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:

**cl470** [*options*] [*filenames*] [**-z** [*link\_options*] [*object files*]]

The `cl470` 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:

**cl470 -z {**-c** | **-cr**} *filenames* [**-o** *name.out*] **-l** *libraryname***

## **Related Documentation From Texas Instruments**

The following books describe the TMS470R1x and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477-8924. When ordering, please identify the book by its title and literature number.

**TMS470R1x Assembly Language Tools User's Guide** (literature number SPNU118) 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 TMS470R1x devices.

**TMS470R1x User's Guide** (literature number SPNU134) describes the TMS470R1x RISC microcontroller, its architecture (including registers), ICEBreaker module, interfaces (memory, coprocessor, and debugger), 16-bit and 32-bit instruction sets, and electrical specifications.

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

## **Related Documentation**

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

**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 Stroustrup, 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***

HP-UX is a trademark of Hewlett-Packard Company.

PC is a trademark of International Business Machines Corp.

Solaris and SunOS are trademarks of Sun Microsystems, Inc.

SPARC is a trademark of SPARC International, Inc., but licensed exclusively to Sun Microsystems, Inc.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Windows and Windows NT are registered trademarks of Microsoft Corporation.

XDS510 is a trademark of Texas Instruments Incorporated.

# Contents

---

---

---

<b>1</b>	<b>Introduction .....</b>	<b>1-1</b>
	<i>Provides an overview of the TMS470R1x software development tools, specifically the compiler.</i>	
1.1	Software Development Tools Overview .....	1-2
1.2	C/C++ Compiler Overview .....	1-5
1.2.1	ANSI/ISO Standard .....	1-5
1.2.2	Output Files .....	1-6
1.2.3	Compiler Interface .....	1-6
1.2.4	Compiler Operation .....	1-7
1.2.5	Utilities .....	1-7
1.3	The Compiler and Code Composer Studio .....	1-8
<b>2</b>	<b>Using the C/C++ Compiler .....</b>	<b>2-1</b>
	<i>Describes how to operate the compiler. Contains instructions for invoking the compiler, which compiles, assembles, and links a source file. Discusses the compiler options, compiler errors, and interlisting.</i>	
2.1	About the Compiler .....	2-2
2.2	Invoking the C/C++ Compiler .....	2-4
2.3	Changing the Compiler's Behavior With Options .....	2-5
2.3.1	Frequently Used Options .....	2-15
2.3.2	Symbolic Debugging and Profiling Options .....	2-18
2.3.3	Specifying Filenames .....	2-19
2.3.4	Changing How the Compiler Interprets Filenames (-fa, -fc, -fg, -fo, and -fp Options) .....	2-20
2.3.5	Changing How the Compiler Interprets and Names Extensions (-ea -ec, -eo, -ep, and -es Options) .....	2-20
2.3.6	Specifying Directories .....	2-21
2.3.7	Options That Control the Assembler .....	2-22
2.3.8	Deprecated Options .....	2-23
2.4	Using Environment Variables .....	2-24
2.4.1	Specifying Directories (C_DIR) .....	2-24
2.4.2	Setting Default Compiler Options (C_OPTION) .....	2-24

2.5	Controlling the Preprocessor .....	2-26
2.5.1	Predefined Macro Names .....	2-26
2.5.2	The Search Path for #include Files .....	2-27
2.5.3	Generating a Preprocessed Listing File (-ppo Option) .....	2-28
2.5.4	Continuing Compilation After Preprocessing (-ppa Option) .....	2-28
2.5.5	Generating a Preprocessed Listing File With Comments (-ppc Option) .....	2-28
2.5.6	Generating a Preprocessed Listing File With Line-Control Information (-ppl Option) .....	2-29
2.5.7	Generating Preprocessed Output for a Make Utility (-ppd Option) .....	2-29
2.5.8	Generating a List of Files Included With the #include Directive (-ppi Option) .....	2-29
2.6	Understanding Diagnostic Messages .....	2-30
2.6.1	Controlling Diagnostics .....	2-32
2.6.2	How You Can Use Diagnostic Suppression Options .....	2-33
2.6.3	Other Messages .....	2-34
2.7	Generating Cross-Reference Listing Information (-px Option) .....	2-35
2.8	Generating a Raw Listing File (-pl Option) .....	2-36
2.9	Using Inline Function Expansion .....	2-38
2.9.1	Inlining Intrinsic Operators .....	2-38
2.9.2	Using the inline Keyword, the -pi Compiler Option, and -O3 Optimization	2-39
2.10	Using Interlist .....	2-40
<b>3</b>	<b>Optimizing Your Code .....</b>	<b>3-1</b>
	<i>Describes how to optimize your C/C++ code, including such features as inlining and loop unrolling. Also describes the types of optimizations that are performed when you use the optimizer.</i>	
3.1	Invoking Optimization .....	3-2
3.2	Performing File-Level Optimization (-O3 Option) .....	3-4
3.2.1	Controlling File-Level Optimizations (-oln Option) .....	3-4
3.2.2	Creating an Optimization Information File (-onn Option) .....	3-5
3.3	Performing Program-Level Optimization (-pm and -O3 Options) .....	3-6
3.3.1	Controlling Program-Level Optimization (-opn Option) .....	3-7
3.3.2	Optimization Considerations When Mixing C/C++ and Assembly .....	3-8
3.4	Using Caution With asm Statements in Optimized Code .....	3-11
3.5	Accessing Aliased Variables in Optimized Code .....	3-12
3.6	Automatic Inline Expansion (-oi Option) .....	3-13
3.7	Using Interlist With Optimization .....	3-14
3.8	Debugging Optimized Code .....	3-16
3.8.1	Debugging Optimized Code (-g, --symdebug:dwarf, --symdebug:coff, and -O Options) .....	3-16
3.8.2	Profiling Optimized Code .....	3-17



3.9	What Kind of Optimization Is Being Performed? .....	3-18
3.9.1	Cost-Based Register Allocation .....	3-19
3.9.2	Alias Disambiguation .....	3-19
3.9.3	Branch Optimizations and Control-Flow Simplification .....	3-19
3.9.4	Data Flow Optimizations .....	3-21
3.9.5	Expression Simplification .....	3-21
3.9.6	Inline Expansion of Runtime-Support Library Functions .....	3-23
3.9.7	Induction Variables and Strength Reduction .....	3-24
3.9.8	Loop-Invariant Code Motion .....	3-24
3.9.9	Loop Rotation .....	3-24
3.9.10	Tail Merging .....	3-24
3.9.11	Autoincrement Addressing .....	3-26
3.9.12	Block Conditionalizing .....	3-27
3.9.13	Epilog Inlining .....	3-28
3.9.14	Removing Comparisons to Zero .....	3-29
3.9.15	Integer Division With Constant Divisor .....	3-29
<b>4</b>	<b>Linking C/C++ Code .....</b>	<b>4-1</b>
	<i>Describes how to link as a stand-alone program or with the compiler shell and how to meet the special requirements of linking C/C++ code.</i>	
4.1	Invoking the Linker (-z Option) .....	4-2
4.1.1	Invoking the Linker As a Separate Step .....	4-2
4.1.2	Invoking the Linker as Part of the Compile Step .....	4-3
4.1.3	Disabling the Linker (-c Option) .....	4-4
4.2	Linker Options .....	4-5
4.3	Controlling the Linking Process .....	4-7
4.3.1	Linking With Run-Time-Support Libraries .....	4-7
4.3.2	Run-Time Initialization .....	4-8
4.3.3	Initialization By the Interrupt Vector .....	4-8
4.3.4	Global Object Constructors .....	4-9
4.3.5	Specifying the Type of Global Variable Initialization .....	4-9
4.3.6	Specifying Where to Allocate Sections in Memory .....	4-10
4.3.7	A Sample Linker Command File .....	4-11
4.3.8	Using Function Subsections (-ms Compiler Option) .....	4-13

<b>5</b>	<b>TMS470R1x C and C++ Languages</b>	<b>5-1</b>
	<i>Discusses the specific characteristics of the TMS470 C/C++ compiler as they relate to the ANSI/ISO C specification.</i>	
5.1	Characteristics of TMS470R1x C	5-2
5.1.1	Identifiers and Constants	5-2
5.1.2	Data Types	5-3
5.1.3	Conversions	5-3
5.1.4	Expressions	5-3
5.1.5	Declarations	5-3
5.1.6	Preprocessor	5-4
5.2	Characteristics of TMS470R1x C++	5-5
5.3	Data Types	5-6
5.4	Keywords	5-7
5.4.1	The const Keyword	5-7
5.4.2	The interrupt Keyword	5-8
5.4.3	The volatile Keyword	5-9
5.5	Register Variables	5-10
5.5.1	Local Register Variables and Parameters	5-10
5.5.2	Global Register Variables	5-11
5.6	The asm Statement	5-13
5.7	Pragma Directives	5-14
5.7.1	The DATA_SECTION Pragma	5-14
5.7.2	The DUAL_STATE Pragma	5-15
5.7.3	The FUNC_EXT_CALLED Pragma	5-16
5.7.4	The INTERRUPT Pragma	5-17
5.7.5	The MUST_ITERATE Pragma	5-18
5.7.6	The TASK Pragma	5-20
5.7.7	The SWI_ALIAS Pragma	5-20
5.7.8	The UNROLL Pragma	5-22
5.8	Generating Linknames	5-23
5.9	Initializing Static and Global Variables	5-24
5.9.1	Initializing Static and Global Variables With the Linker	5-24
5.9.2	Initializing Static and Global Variables With the Const Type Qualifier	5-25
5.10	Changing the Language Mode (-pk, -pr, and -ps Options)	5-26
5.10.1	Compatibility With K&R C (-pk Option)	5-26
5.10.2	Enabling Strict ANSI/ISO Mode and Relaxed ANSI/ISO Mode (-ps and -pr Options)	5-28
5.10.3	Enabling Embedded C++ Mode (-pe Option)	5-28
5.11	Compiler Limits	5-29

## 6 Run-Time Environment ..... 6-1

*Describes how the compiler uses the TMS470 architecture. Discusses memory, register, and function-calling conventions, object representation, system initialization, and dual-state interworking. Provides information for interfacing assembly language to C/C++ programs.*

6.1	Memory Model .....	6-2
6.1.1	Sections .....	6-2
6.1.2	C/C++ Software Stack .....	6-4
6.1.3	Dynamic Memory Allocation .....	6-5
6.1.4	Initialization of Variables .....	6-5
6.2	Object Representation .....	6-6
6.2.1	Data Type Storage .....	6-6
6.2.2	Bit Fields .....	6-11
6.2.3	Character String Constants .....	6-12
6.3	Register Conventions .....	6-13
6.4	Function Structure and Calling Conventions .....	6-15
6.4.1	How a Function Makes a Call .....	6-17
6.4.2	How a Called Function Responds .....	6-17
6.4.3	Accessing Arguments and Local Variables .....	6-19
6.4.4	Generating Long Calls (–ml Option) in 16-bit Mode .....	6-19
6.5	Interfacing C/C++ With Assembly Language .....	6-20
6.5.1	Using Assembly Language Modules with C/C++ Code .....	6-20
6.5.2	Accessing Assembly Language Variables From C .....	6-23
6.5.3	Using Inline Assembly Language .....	6-25
6.5.4	Modifying Compiler Output .....	6-25
6.6	Interrupt Handling .....	6-26
6.6.1	Saving Registers During Interrupts .....	6-26
6.6.2	Using C/C++ Interrupt Routines .....	6-26
6.6.3	Using Assembly Language Interrupt Routines That Call C/C++ Functions .....	6-27
6.6.4	How to Map Interrupt Routines to Interrupt Vectors .....	6-27
6.6.5	Using Software Interrupts .....	6-28
6.6.6	Other Interrupt Information .....	6-29
6.7	Intrinsic Run-Time-Support Arithmetic and Conversion Routines .....	6-30
6.7.1	Naming Conventions .....	6-30
6.8	Built-In Functions .....	6-34
6.9	System Initialization .....	6-35
6.9.1	Run-Time Stack .....	6-35
6.9.2	Automatic Initialization of Variables .....	6-36
6.9.3	Global Constructors .....	6-36
6.9.4	Initialization Tables .....	6-37
6.9.5	Autoinitialization of Variables at Run Time .....	6-39
6.9.6	Autoinitialization of Variables at Load Time .....	6-40
6.10	Dual-State Interworking .....	6-41
6.10.1	Level of Dual-State Support .....	6-41
6.10.2	Implementation .....	6-43

<b>7</b>	<b>Run-Time-Support Functions</b>	<b>7-1</b>
	<i>Describes the libraries and header files included with the C/C++ compiler, as well as the macros, functions, and types that they declare. Summarizes the run-time-support functions according to category (header) and provides an alphabetical reference for the run-time-support functions.</i>	
7.1	Libraries	7-2
7.1.1	Nonstandard Header Files in rtsc.src and rtscpp.src	7-2
7.1.2	Modifying a Library Function	7-3
7.1.3	Building a Library With Different Options	7-3
7.2	The C I/O Functions	7-4
7.2.1	Overview of Low-Level I/O Implementation	7-5
7.2.2	Adding A Device for C I/O	7-14
7.3	Header Files	7-16
7.3.1	Diagnostic Messages (assert.h/cassert)	7-17
7.3.2	Character-Typing and Conversion (ctype.h/cctype)	7-18
7.3.3	Error Reporting (errno.h/cerrno)	7-18
7.3.4	Low-Level Input/Output Functions (file.h)	7-18
7.3.5	Limits (float.h/cfloat and limits.h/climits)	7-19
7.3.6	Format Conversion of Integer Types (inttypes.h)	7-21
7.3.7	Alternative Spellings (iso646.h/ciso646)	7-22
7.3.8	Function Calls as near or far (linkage.h)	7-22
7.3.9	Floating-Point Math (math.h/cmath)	7-22
7.3.10	Nonlocal Jumps (setjmp.h/csetjmp)	7-23
7.3.11	Variable Arguments (stdarg.h/cstdarg)	7-23
7.3.12	Standard Definitions (stddef.h/cstddef)	7-24
7.3.13	Integer Types (stdint.h)	7-24
7.3.14	Input/Output Functions (stdio.h/cstdio)	7-25
7.3.15	General Utilities (stdlib.h/cstdlib)	7-26
7.3.16	String Functions (string.h/cstring)	7-26
7.3.17	Time Functions (time.h/ctime)	7-27
7.3.18	Exception Handling (exception and stdexcept)	7-28
7.3.19	Dynamic Memory Management (new)	7-28
7.3.20	Run-Time Type Information (typeinfo)	7-28
7.4	Summary of Run-Time-Support Functions and Macros	7-29
7.5	Description of Run-Time-Support Functions and Macros	7-39
<b>8</b>	<b>Library-Build Utility</b>	<b>8-1</b>
	<i>Describes the utility that custom-makes run-time-support libraries for the options used to compile code. You can use this utility to install header files in a directory and to create custom libraries from source archives.</i>	
8.1	Invoking the Library-Build Utility	8-2
8.2	Library-Build Utility Options	8-3
8.3	Options Summary	8-4

---

<b>9</b>	<b>C++ Name Demangler</b> .....	<b>9-1</b>
	<i>Describes the C++ name demangler and tells you how to invoke and use it.</i>	
9.1	Invoking the C++ Name Demangler .....	9-2
9.2	C++ Name Demangler Options .....	9-2
9.3	Sample Usage of the C++ Name Demangler .....	9-3
<b>10</b>	<b>Static Stack Depth Profiler</b> .....	<b>10-1</b>
	<i>Describes the static stack depth profiler, how to invoke it, and how to read the statistics listing.</i>	
10.1	Invoking the Static Stack Depth Profiler .....	10-2
10.2	Stack Depth Statistics Listing .....	10-3
10.3	Dependencies and Limitations .....	10-5
10.4	User Input Mechanisms .....	10-6
10.5	Configuration File Specification .....	10-7
	10.5.1 Specify Indirect Calls .....	10-7
	10.5.2 Specifying Reentrant Procedures .....	10-8
	10.5.3 Specifying Interrupt Service Routines .....	10-9
	10.5.4 Other Features .....	10-10
	10.5.5 Tail Calls .....	10-11
10.6	Profiler Generated Warnings .....	10-12
10.7	Run-Time Support for Stack Depth Profiling .....	10-13
<b>11</b>	<b>Glossary</b> .....	<b>A-1</b>
	<i>Defines terms and acronyms used in this book.</i>	

# Figures

---

---

---

1-1	TMS470R1x Software Development Flow .....	1-2
2-1	The Compiler Overview .....	2-3
3-1	Compiling a C/C++ Program With Optimization .....	3-2
6-1	Char and Short Data Storage Format .....	6-7
6-2	32-Bit Data Storage Format .....	6-8
6-3	Double-Precision Floating-Point Data in Register Pairs .....	6-9
6-4	Bit-Field Packing in Big-Endian and Little-Endian Formats .....	6-11
6-5	Use of the Stack During a Function Call .....	6-16
6-6	Format of Initialization Records in the .cinit Section .....	6-37
6-7	Format of the .pinit Section .....	6-38
6-8	Autoinitialization at Run Time .....	6-39
6-9	Autoinitialization at Load Time .....	6-40
7-1	Interaction of Data Structures in I/O Functions .....	7-5
7-2	The First Three Streams in the Stream Table .....	7-6
3-1	Application Call Trees .....	10-3

# Tables

2-1	Compiler Options Summary .....	2-6
2-2	Compiler Backwards-Compatibility Options Summary .....	2-23
2-3	Predefined Macro Names .....	2-26
2-4	Raw Listing File Identifiers .....	2-36
2-5	Raw Listing File Diagnostic Identifiers .....	2-36
3-1	Options That You Can Use With -O3 .....	3-4
3-2	Selecting a Level for the -ol Option .....	3-4
3-3	Selecting a Level for the -on Option .....	3-5
3-4	Selecting a Level for the -op Option .....	3-7
3-5	Special Considerations When Using the -op Option .....	3-8
4-1	Sections Created by the Compiler .....	4-10
5-1	TMS470R1x C Data Types .....	5-6
6-1	Summary of Sections and Memory Placement .....	6-3
6-2	Data Representation in Registers and Memory .....	6-6
6-3	How Register Types Are Affected by the Conventions .....	6-13
6-4	Register Usage and Preservation Conventions .....	6-14
6-5	Naming Convention Prefixes .....	6-30
6-6	Summary of Run-Time-Support Arithmetic and Conversion Functions .....	6-31
6-7	Run-Time-Support Function Register Usage Conventions .....	6-32
6-8	Selecting a Level of Dual-State Support .....	6-42
7-1	Macros That Supply Integer Type Range Limits (limits.h/climits) .....	7-19
7-2	Macros That Supply Floating-Point Range Limits (float.h/cfloat) .....	7-20
7-3	Summary of Run-Time-Support Functions and Macros .....	7-30
8-1	Summary of Options and Their Effects .....	8-4

# Examples

---

---

---

2-1	An Interlisted Assembly Language File .....	2-40
3-1	Listing for Compilation With the -o2 and -s Options .....	3-14
3-2	Listing for Compilation With the -o2, -os, and -ss Options .....	3-15
3-3	Control-Flow Simplification and Copy Propagation .....	3-20
3-4	Data Flow Optimizations and Expression Simplification .....	3-22
3-5	Inline Function Expansion .....	3-23
3-6	Tail Merging .....	3-25
3-7	Autoincrement Addressing, Loop-Invariant Code Motion, and Strength Reduction ....	3-26
3-8	Block Conditionalizing .....	3-27
3-9	Epilog Inlining .....	3-28
3-10	Removing Comparisons to Zero .....	3-29
4-1	Linker Command File .....	4-12
5-1	Using the DATA_SECTION Pragma .....	5-15
5-2	Using the SWI_ALIAS Pragma .....	5-21
6-1	An Assembly Language Function .....	6-22
6-2	Accessing an Assembly Language Variable From C .....	6-23
6-3	Accessing an Assembly Language Constant From C .....	6-24
6-4	Initialization Variables and Initialization Table .....	6-38
6-5	Code Compiled for 16-BIS State: sum( ) .....	6-45
6-6	Code Compiled for 32-BIS State: max( ) .....	6-46
9-1	Name Mangling .....	9-3
9-2	Result After Running the C++ Name Demangler Utility .....	9-4
10-1	Profile of the Application Call Trees Figure .....	10-4
10-2	An Assembly Function .....	10-6
10-3	Specifying Indirect Calls .....	10-7



# Notes

Function Inlining Can Greatly Increase Code Size .....	2-38
Using the <code>-pi</code> Option With <code>-O3</code> Optimizations .....	2-39
Compiling Files With the <code>-pm</code> and <code>-k</code> Options .....	3-6
<code>-O3</code> Optimization and Inlining .....	3-13
Inlining and Code Size .....	3-13
Impact on Performance and Code Size .....	3-15
Symbolic Debugging Options Affect Performance and Code Size .....	3-16
Profile Points .....	3-17
Order of Processing Arguments in the Linker .....	4-4
The <code>_c_int00</code> Symbol .....	4-8
Boot Loader .....	4-10
Avoid Disrupting the C/C++ Environment With <code>asm</code> Statements .....	5-13
The Linker Defines the Memory Map .....	6-2
Using the <code>asm</code> Statement .....	6-25
Prefixing C/C++ Interrupt Routines .....	6-27
Use Unique Function Names .....	7-14
Writing Your Own Clock Function .....	7-28
Writing Your Own Clock Function .....	7-47
The <code>getenv</code> Function Is Target-System Specific .....	7-60
No Previously Allocated Objects are Available After <code>init</code> .....	7-69
No Previously Allocated Objects are Available After <code>init</code> .....	7-69
The <code>time</code> Function Is Target-System Specific .....	7-95

---

## Introduction

---

---

---

The TMS470R1x is supported by a set of software development tools, which includes an optimizing C/C++ compiler, an assembler, a linker, and assorted utilities.

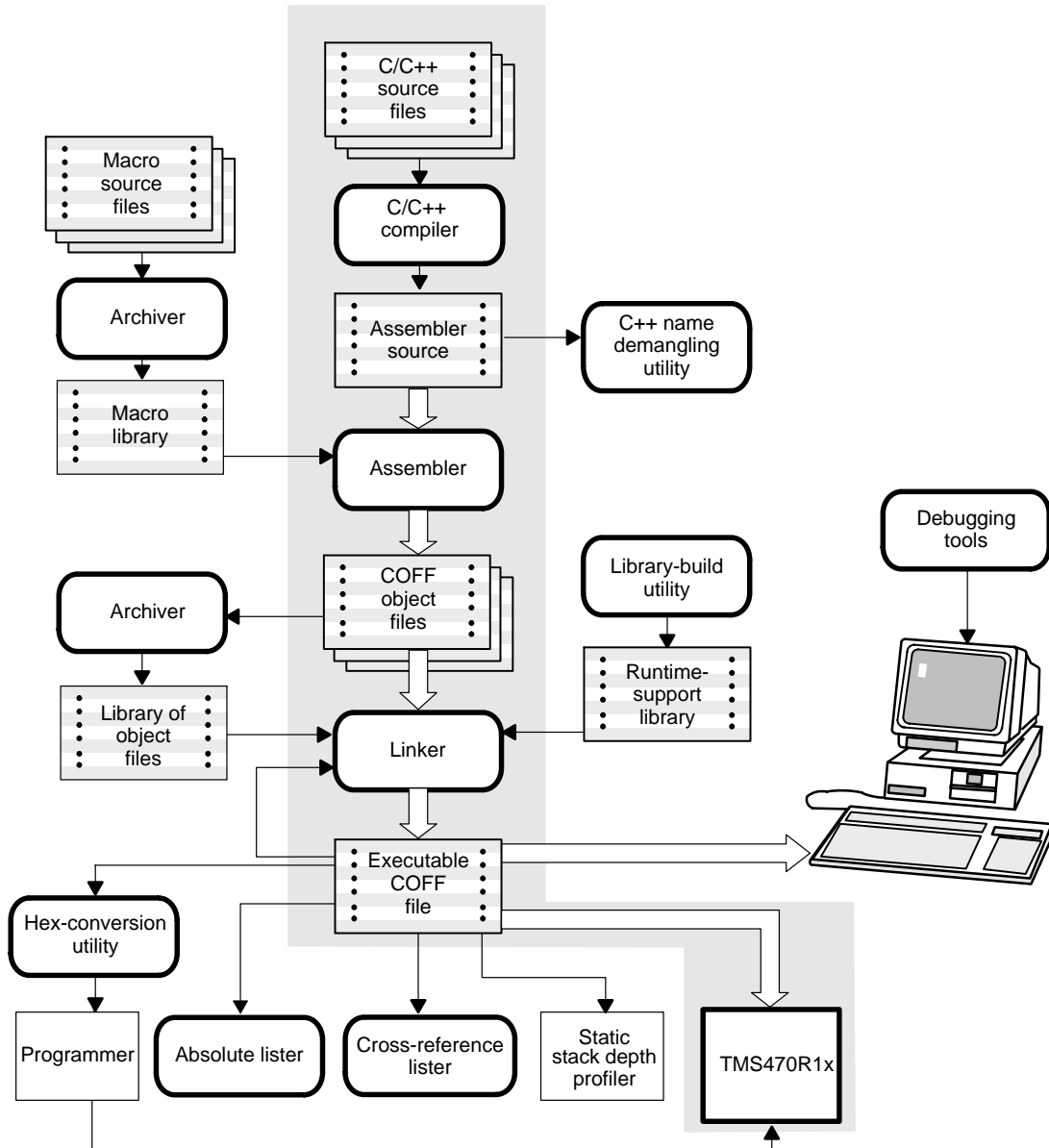
This chapter provides an overview of these tools and introduces the features of the optimizing C/C++ compiler. The assembler and linker are discussed in detail in the *TMS470R1x Assembly Language Tools User's Guide*.

Topic	Page
1.1 Software Development Tools Overview .....	1-2
1.2 C/C++ Compiler Overview .....	1-5
1.3 The Compiler and Code Composer Studio .....	1-8

## 1.1 Software Development Tools Overview

Figure 1–1 illustrates the TMS470 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. TMS470R1x 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 TMS470 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 *TMS470R1x 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 *TMS470R1x 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 *TMS470R1x Assembly Language Tools User's Guide* explains how to use the archiver.

- ❑ You can use the **library-build utility** to build your own customized run-time-support library (see Chapter 8, *Library-Build Utility*). Standard run-time-support library functions are provided as source code in `rtsc.src`. The library object code for the run-time-support functions is compiled for 16-bit processing in `rtsc_16.lib` and for 32-bit processing in `rtsc_32.lib`.

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 TMS470 compiler. See Chapter 7, *Run-Time-Support Functions*, for more information.

- ❑ The TMS470 debugger accepts executable COFF files as input, but most EPROM programmers do not. The **hex-conversion utility** converts a COFF object file into TI-Tagged, ASCII-hex, Intel, Motorola-S, or Tektronix object format. The converted file can be downloaded to an EPROM programmer. The *TMS470R1x 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 *TMS470R1x 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 *TMS470R1x 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 **static depth profiler** provides information about the maximum stack depth requirements of an application based on the static information available in the linker output file.
- ❑ The main product of this development process is a module that can be executed on a TMS470R1x device. You can use one of several debugging tools to refine and correct your code. Available products include:
  - An instruction-accurate and clock-accurate software simulator
  - A fast simulator for testing code functionality
  - An extended development system (XDS510™) emulator

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

## 1.2 C/C++ Compiler Overview

The TMS470 C/C++ compiler is a full-featured optimizing compiler that translates standard ANSI/ISO C/C++ programs into TMS470 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 TMS470 C/C++ compiler fully 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.

#### ☐ ANSI/ISO-standard C++

The TMS470 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 *TMS470R1x 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 2-26.

### ☐ **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 TMS470-specific optimizations take advantage of the features specific to the TMS470 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 utility** is a significant feature of the compiler utilities. The library-build utility 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 Studio

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

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

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

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

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

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

# Using the C/C++ Compiler

The compiler translates your source program into code that the TMS470R1x™ 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, cl470. This chapter provides a complete description of how to use cl470 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 .....	2-2
2.2 Invoking the C/C++ Compiler .....	2-4
2.3 Changing the Compiler's Behavior With Options .....	2-5
2.4 Using Environment Variables .....	2-24
2.5 Controlling the Preprocessor .....	2-26
2.6 Understanding Diagnostic Messages .....	2-30
2.7 Generating Cross-Reference Listing Information (-px Option) .....	2-35
2.8 Generating a Raw Listing File (-pl Option) .....	2-36
2.9 Using Inline Function Expansion .....	2-38
2.10 Using the Interlist Utility .....	2-40

## 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:

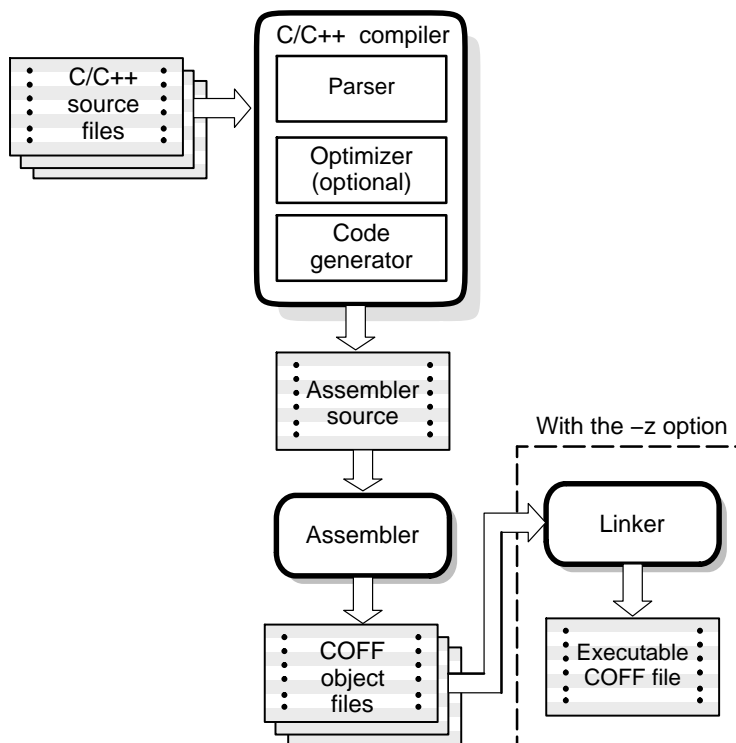
- ❑ The **compiler**, which includes the parser and optimizer, accepts C/C++ source code and produces TMS470 assembly language source code.

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

- ❑ The **assembler** generates a COFF object file.
- ❑ The **linker** 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 linker by using the `-z` compiler option. Figure 2–1 illustrates the path the compiler takes with and without using the linker.

Figure 2–1. The Compiler Overview



For a complete description of the assembler and the linker, see the *TMS470R1x Assembly Language Tools User's Guide*.

## 2.2 Invoking the C/C++ Compiler

To invoke the compiler, enter:

**cl470** [*options*] [*filenames*] [**-z** [*link\_options*] [*object files*]]

<b>cl470</b>	Command that runs the compiler and the assembler
<i>options</i>	Options that affect the way the input files are processed (the options are listed in Table 2–1 on page 2-6)
<i>filenames</i>	One or more C/C++ source files, assembly source files, or object files
<b>-z</b>	Option that invokes the linker. See Chapter 4, <i>Linking C/C++ Code</i> , for more information about invoking the linker.
<i>link_options</i>	Options that control the linking process
<i>object files</i>	Name of the additional object files for the linking process

The arguments to cl470 are of three types: compiler options, linker options, and files. The **-z** linker option is the signal that linking is to be performed. If the **-z** linker option is used, compiler options must precede the **-z** linker options, and other linker options must follow the **-z** linker option. Source code filenames must be placed before the **-z** linker option. Additional object file filenames may be placed after the **-z** 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:

```
cl470 -q symtab.c file.c seek.asm-z -llink.cmd -lrts470.lib
```

## 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 `cl470` 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 either single letters or 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 `-U=name`. Although not recommended, you can separate the option and the parameter with or without a space, as in `-U name` or `-Uname`.
- ☐ 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 `-O=3`. Although not recommended, you can specify the parameter directly after the option, as in `-O3`. No space is allowed between the option and the optional parameter, so `-O 3` is not accepted.
- ☐ Files and options except the `-z` option can occur in any order. The `-z` option must follow all other compiler options and precede any linker options.

You can define default options for the compiler by using the `C_OPTION` environment variable. For more information on the `C_OPTION` environment variable, see section 2.4.2, *Setting Default Compiler Options (C\_OPTION and 470\_C\_OPTION)*, on page 2-24.

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

For an online summary of the options, enter **cl470** with no parameters on the command line.

Table 2–1. Compiler Options Summary

(a) Options that control the compiler

Option	Effect	Page
<code>-@=filename</code>	Interprets contents of <i>filename</i> as an extension to the command line	2-15
<code>-b</code>	Generates auxiliary information file	2-15
<code>-c</code>	Disables linking (negates <code>-z</code> )	2-15, 4-4
<code>-D=name[=def]</code>	Predefines <i>name</i>	2-15
<code>-I=directory</code>	Defines <code>#include</code> search path	2-15, 2-27
<code>-h</code>	Shows help screen	--
<code>-k</code>	Keeps the assembly language (.asm) file	2-15
<code>-n</code>	Compiles only	2-11
<code>-q</code>	Suppresses progress messages (quiet)	2-16
<code>-s</code>	Interlists optimizer comments (if available) and assembly statements; otherwise, interlists C source and assembly source statements	2-17, 2-40
<code>-ss</code>	Interlists C source and assembly statements	2-17, 3-14
<code>-U=name</code>	Undefines <i>name</i>	2-17
<code>--verbose</code>	Displays banner and copyright information	2-17
<code>--version</code>	Prints version number for each tool	2-18
<code>-z[=filename]</code>	Enables linking using options specified after <code>-z</code>	2-18



Table 2–1. Compiler Options Summary (Continued)

*(b) Options that control symbolic debugging and profiling*

Option	Effect	Page
-g	Enables symbolic debugging (equivalent to <code>--symdebug:dwarf</code> )	2-18, 3-16
--profile:breakpt	Enables breakpoint-based profiling	2-18
--profile:power	Enables power profiling	2-18
--symdebug:coff	Enables symbolic debugging using the alternate STABS debugging format	2-19, 3-16
--symdebug:dwarf	Enables symbolic debugging using the DWARF debugging format (equivalent to -g)	2-18, 3-16
--symdebug:none	Disables all symbolic debugging	2-19
--symdebug:skeletal	Enables minimal symbolic debugging that does not hinder optimizations (default behavior)	2-19

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

Option	Effect	Page
-ea=.ext	Sets default extension for assembly files	2-20
-eo=.ext	Sets default extension for object files	2-20
-ec=.ext	Sets default extension for C source files	2-20
-ep=.=ext	Sets default extension for C++ source files	2-20
-es=.ext	Sets default extension for assembly listing files	2-20

Table 2–1. Compiler Options Summary (Continued)

*(d) Options that specify files*

Option	Effect	Page
<code>-fa=filename</code>	Identifies <i>filename</i> as an assembly file, regardless of its extension. By default, the compiler treats <code>.asm</code> or <code>.s*</code> (extension begins with <code>s</code> ) as an assembly file.	2-20
<code>-fc=filename</code>	Identifies <i>filename</i> as a C file, regardless of its extension. By default, the compiler treats <code>.c</code> or no extension as a C file.	2-20
<code>-fg</code>	Causes all C files to be treated as C++ files.	2-20
<code>-fo=filename</code>	Identifies <i>filename</i> as an object file, regardless of its extension. By default, the compiler treats <code>.o*</code> as an object file.	2-20
<code>-fp=filename</code>	Identifies <i>filename</i> as a C++ file, regardless of its extension. By default, the compiler treats <code>.C</code> , <code>.cpp</code> , <code>.cc</code> , or <code>.cxx</code> as a C++ file. <sup>†</sup>	2-20

*(e) Options that specify directories*

Option	Effect	Page
<code>-fb=directory</code>	Specifies absolute listing file directory	2-21
<code>-ff=directory</code>	Specifies listing/xref file directory	2-21
<code>-fr=directory</code>	Specifies object file directory	2-21
<code>-fs=directory</code>	Specifies assembly file directory	2-21
<code>-ft=directory</code>	Specifies temporary file directory	2-21

Table 2-1. Compiler Options Summary (Continued)

(f) Options that control parsing

Option	Effect	Page
--exceptions	Enables C++ exception handling	
-pc	Multibyte character support	--
-pe	Enables embedded C++ mode	5-28
-pi	Disables definition-controlled inlining (but -O3 optimizations still perform automatic inlining)	2-39
-pk	Allows K&R compatibility	5-26
-pl	Outputs raw listing information	2-36
-pm	Combines source files to perform program-level optimization	3-6
-pn	Disable intrinsic functions	--
-pr	Enables relaxed mode; ignores strict ANSI violations	5-28
-ps	Enables strict ANSI mode (for C/C++, not K&R C)	5-28
-px	Generates a cross-reference listing file	2-35
-rtti	Enables run-time type information (RTTI), which allows the type of an object to be determined at run time.	5-5
--static_template_instantiation	Instantiates all template entities with internal linkage	

Table 2–1. Compiler Options Summary (Continued)

*(g) Parser options that control preprocessing*

Option	Effect	Page
-ppa	Continues compilation after preprocessing	2-28
-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
-ppd[=file]	Performs preprocessing only, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility	2-29
-ppi[=file]	Performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive	2-29
-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	2-29
-ppo	Performs preprocessing only. Writes preprocessed output to a file with the same name as the input but with a .pp extension	2-28

*(h) Parser options that control diagnostics*

Option	Effect	Page
-pdel=num	Sets the error limit to <i>num</i> . The compiler abandons compiling after this number of errors. (The default is 100.)	2-32
-pden	Displays a diagnostic's identifiers along with its text	2-32
-pdf	Generates a diagnostics information file	2-32
-pdr	Issues remarks (nonserious warnings)	2-32
-pds=num	Suppresses the diagnostic identified by <i>num</i>	2-32
-pdse=num	Categorizes the diagnostic identified by <i>num</i> as an error	2-32
-pdsr=num	Categorizes the diagnostic identified by <i>num</i> as a remark	2-32
-pds=warning	Categorizes the diagnostic identified by <i>num</i> as a warning	2-30
-pdv	Provides verbose diagnostics that display the original source with line-wrap	2-33
-pdw	Suppresses warning diagnostics (errors are still issued)	2-33

Table 2–1. Compiler Options Summary (Continued)

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

Option	Effect	Page
<code>-ma</code>	Assumes variables are aliased	2-16
<code>-ab=num</code>	Specifies length of maximum branch chain for branch chaining optimization	--
<code>-abi={ti_arm9_abi tiabi}</code>	Specifies application binary interface	--
<code>--align_structs=bytecount</code>	Forces alignment of structures to <i>bytecount</i> bytes.	--
<code>--code_state={16 32}</code>	Designates code state as 16-bit (thumb), or default 32-bit (arm)	--
<code>--endian={big little}</code>	Designates endianness, default is big	
<code>-mc</code>	Changes variables of type <code>char</code> from unsigned to signed	2-16
<code>-md</code>	Disables dual-state interworking support	6-43
<code>-me</code>	Produces object code in little-endian format	2-16
<code>-mf</code>	Optimizes for speed over space	2-16
<code>-ml</code>	Use long calls (requires <code>-md</code> and <code>-mt</code> )	6-19
<code>-mn</code>	Enables optimizations disabled by <code>-g</code>	2-16
<code>-mo</code>	Enables dynamic stack overflow checking	--
<code>-ms</code>	Turns on function subsections	4-13
<code>-mt</code>	Generates 16-bit code	2-16
<code>-mv={4 5e 6}</code>	Selects processor version: ARM4, ARM5e, ARM6; default is ARM4	--
<code>--plain_char={signed unsigned}</code>	Specifies how to treat plain chars, default is unsigned	--
<code>-r={r5 r6 r9}</code>	Disallows use of <code>rx=[5,6,9]</code> by the compiler	--
<code>--small-enum</code>	Uses the smallest possible size for the enumeration type	2-16
<code>--use_dead_funcs_list [=fname]</code>	Places each function listed in the file in a separate section	--

Table 2–1. Compiler Options Summary (Continued)

(j) Options that control optimization

Option	Effect	Page
-O0	Optimizes register usage	3-2
-O1	Uses -O0 optimizations and optimizes locally	3-2
-O2 or -O	Uses -O1 optimizations and optimizes globally	3-3
-O3	Uses -O2 optimizations and optimizes file	3-3, 3-4
-oi=size	Sets automatic inlining size (-O3 only)	3-13
-ol0 (-oL0)	Informs the optimizer that your file alters a standard library function	3-4
-ol1 (-oL1)	Informs the optimizer that your file declares a standard library function	3-4
-ol2 (-oL2)	Informs the optimizer that your file does not declare or alter library functions. Overrides the -ol0 and -ol1 options	3-4
-on0	Disables optimizer information file	3-5
-on1	Produces optimizer information file	3-5
-on2	Produces verbose optimizer information file	3-5
-op0	Specifies that the module contains functions and variables that are called or modified from outside the source code provided to the compiler	3-7
-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
-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
-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
-os	Interlists optimizer comments with assembly statements	3-14

Table 2–1. Compiler Options Summary (Continued)

*(k) Options that control the assembler*

Option	Effect	Page
-aa	Enables absolute listing	2-22
-ac	Makes case significant in assembly source files	2-22
-ad= <i>name</i>	Sets the <i>name</i> symbol	2-22
-ahc= <i>filename</i>	Copies the specified file for the assembly module	2-22
-ahi= <i>filename</i>	Includes the file for the assembly module	2-22
-al	Generates an assembly listing file	2-22
-apd	Performs preprocessing; lists only assembly dependencies	2-22
-api	Performs preprocessing; lists only included #include files	2-22
-as	Puts labels in the symbol table	2-22
-au= <i>name</i>	Undefines the predefined constant <i>name</i>	2-22
-ax	Generates the cross-reference file	2-22

*(l) Options that control the linker*

Options	Effect	Page
-a	Generates absolute output	4-5
-abs	Produces an absolute listing file.	2-15
-ar	Generates relocatable output	4-5
--args= <i>size</i>	Allocates memory to be used by the loader to pass arguments	4-5
-b	Disables merge of symbolic debugging information	4-5
-c	Autoinitializes variables at run time	4-5
-cr	Autoinitializes variables at load time	4-5
-e= <i>global_symbol</i>	Defines entry point	4-5
-f= <i>fill_value</i>	Defines fill value	4-5
-g= <i>global_symbol</i>	Keeps a <i>global_symbol</i> global (overrides -h)	4-5
-h	Makes global symbols static	4-5
-heap= <i>size</i>	Sets heap size (in bytes)	4-5

Table 2–1. Compiler Options Summary (Continued)

(l) Options that control the linker (continued)

Options	Effect	Page
<code>-I=directory</code>	Defines library search path	4-5
<code>-j</code>	Disables conditional linking	4-5
<code>-l=filename</code>	Supplies library name	4-6
<code>--large_model</code>	Generates far call trampolines	
<code>-m=filename</code>	Names the map file	4-6
<code>-o=filename</code>	Names the output file	4-6
<code>-opt=options</code>	Perform link-time optimization	
<code>-priority</code>	Provides an alternate search mechanism for libraries	4-6
<code>-r</code>	Retains relocation entries in the output module	4-6
<code>-s</code>	Strips symbol table	4-6
<code>-stack=size</code>	Sets stack size (in bytes)	4-6
<code>-U=symbol</code>	Undefines <i>symbol</i>	4-6
<code>-w</code>	Warns if an unspecified output section is created	4-6
<code>-wr</code>	Warns if a symbol is redefined	
<code>-x</code>	Forces rereading of libraries	4-6



## 2.3.1 Frequently Used Options

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

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

Within the command file, filenames or option parameters containing embedded spaces or hyphens must be surrounded by quotation marks. For example: "this-file.obj"
- abs**                  Generates an absolute listing file when used after the -z option. Note that you must use the -O option (after -z) to specify the .out file for the absolute lister, even if you use a linker command file that already uses -O.
- b**                    Generates an auxiliary information file that you can refer to for information about stack size and function calls. The filename is the C/C++ source filename with a .aux extension.
- c**                    Suppresses the linker and overrides the -z option, which specifies linking. Use this option when you have -z specified in the C\_OPTION environment variable and you do not want to link. For more information, see section 4.1.3, *Disabling the Linker (-c Compiler Option)*, on page 4-4.
- D=*name*[=*def*]**    Predefines the constant *name* for the preprocessor. This is equivalent to inserting `#define name def` at the top of each C/C++ source file. If the optional [*def*] is omitted, the *name* is set to 1.
- I=*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 -I options with spaces. If you do not specify a directory name, the preprocessor ignores the -I option. For more information, see section 2.5.2.1, *Changing the #include File Search Path With the -I Option*, on page 2-27.
- k**                    Keeps the assembly language output from the compiler. Normally, the compiler deletes the output assembly language file after assembly is complete.

<b>-ma</b>	Assumes that variables are aliased. The compiler assumes that pointers may alias (point to) named variables. Therefore, it disables register optimizations when an assignment is made through a pointer when the compiler determines that there may be another pointer pointing to the same object.
<b>-mc</b>	Changes C/C++ variables of type char from unsigned to signed. By default, char variables are unsigned.
<b>-me</b>	Produces code in little-endian format. By default, big-endian code is produced.
<b>-mf</b>	Optimizes your code for speed over size. By default, the TMS470 optimizer attempts to reduce the size of your code at the expense of speed.
<b>-mn</b>	Reenables the optimizations disabled by the -g option. If you use the -g option, many code generator optimizations are disabled because they disrupt the debugger. Therefore, if you use the -mn option, portions of the debugger's functionality will be unreliable.
<b>-mt</b>	Generates 16-bit code. By default, 32-bit code is generated.
<b>-n</b>	Compiles only. The specified source files are compiled, but not assembled or linked. This option overrides -z. The output is assembly language output from the compiler.
<b>-q</b>	Suppresses banners and progress information from all the tools. Only source filenames and error messages are output.
<b>--small-enum</b>	<p>By default, the TMS470 compiler uses 32 bits for every <i>enum</i>. When you use the --small-enum option, the smallest possible byte size for the enumeration type is used. For example, <code>enum example_enum {first = -128, second = 0, third = 127}</code> will use only one byte instead of 32 bits when the --small-enum option is used. Similarly, <code>enum a_short_enum {bottom = -32768, middle = 0, top = 32767}</code> will fit into two bytes instead on four.</p> <p>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 can not be detected until run time.</p>

- s** Invokes the interlist feature, which interweaves optimizer comments *or* C/C++ source with assembly source. If the optimizer is invoked (*-On* 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 *-s* option implies the *-k* option. For more information about using the interlist feature with the optimizer, see section 3.7, *Using Interlist With Optimization*, on page 3-14.
- ss** Invokes the interlist feature, which interweaves original C/C++ source with compiler-generated assembly language. If the optimizer is invoked (*-On* option) along with this option, your code might be reorganized substantially. For more information, see section 2.10, *Using Interlist*, on page 2-40.
- U=name** Undefined the predefined constant *name*. This option overrides any *-D* options for the specified constant.
- verbose** Displays the banner while compiling. This includes the compiler version number and copyright information. For example:

```
cl470 --verbose file.c
```

```
TMS470 C/C++ Compiler           Version 3.0
Tools Copyright (c) 1996-2003 Texas Instruments Incorporated
"file.c"    ++> main
```

**--version** Prints the version number for each tool in the compiler. No compiling occurs. For example:

```
cl470 --version
```

```
TMS470 C/C++ Compiler           Version 3.00
```

```
Build Number 1DK2N-8A0FEEH-UASQC-VAV-DZAZE_T_QQ_1S
```

```
TMS470 C/C++ Parser             Version 3.00
```

```
Build Number 1DK2N-8A0FEEH-UASQC-VAV-DZAZE_T_QQ_1S
```

```
TMS470 C/C++ Optimizer          Version 3.00
```

```
Build Number 1DK2N-8A0FEEH-UASQC-VAV-DZAZE_T_QQ_1S
```

```
TMS470 C/C++ Codegen            Version 3.00
```

```
Build Number 1DK2N-8A0FEEH-UASQC-VAV-DZAZE_T_QQ_1S
```

```
TMS470 COFF Assembler           Version 3.00
```

```
Build Number 1DK2N-8A0FEEH-UASQC-VAV-DZAZE_T_QQ_1S
```

```
TMS470 COFF Linker              Version 3.00
```

```
Build Number 1DK2N-8A0FEEH-UASQC-VAV-DZAZE_T_QQ_1S
```

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

### 2.3.2 Symbolic Debugging and Profiling Options

Following are options 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 **-O** option to maximize the amount of optimization that is compatible with debugging (see section 3.8, *Debugging Optimized Code*, on page 3-16).

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

**--profile:breakpt** Disables optimizations that would cause incorrect behavior when using a breakpoint-based profiler.

**--profile:power** Enables power profiling which produces instrument code for the power profiler.

- symdebug:coff** Enables symbolic debugging using the alternate STABS debugging format. This may be necessary to allow debugging with older debuggers or custom tools, which do not read the DWARF format.
- symdebug:none** Disables all symbolic debugging output. This option is not recommended; it prevents debugging and most performance analysis capabilities.
- symdebug:skeletal** Generates as much symbolic debugging information as possible without hindering optimization. Generally, this consists of global-scope information only. This option reflects the default behavior of the compiler.

See section 2.3.8 on page 2-23 for deprecated symbolic debugging options.

### 2.3.3 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 <sup>†</sup>	C++ source
.asm, .abs, or .s* (extension begins with s)	Assembly source
.obj	Object

<sup>†</sup> 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 2.3.4. For information about how you can alter the way the compiler interprets and names the extensions of assembly source and object files, see section 2.3.5.

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:

```
c1470 *.c
```

### 2.3.4 Changing How the Compiler Interprets Filenames (`-fa`, `-fc`, `-fg`, `-fo`, and `-fp` Options)

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 `-fa`, `-fc`, `-fo`, and `-fp` options to specify the type of file. You can insert an optional space between the option and the filename. Select the appropriate option for the type of file you want to specify:

<code>-fa=filename</code>	for an assembly language source file
<code>-fc=filename</code>	for a C source file
<code>-fo=filename</code>	for an object file
<code>-fp=filename</code>	for a C++ source file

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

```
cl470 -fc=file.s -fa assy
```

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

The `-fg` option causes the compiler to process C files as C++ files. By default, the compiler treats files with a `.c` extension as C files. See section 2.3.3 on page 2-19 for more information about filename extension conventions.

### 2.3.5 Changing How the Compiler Interprets and Names Extensions (`-ea`, `-ec`, `-eo`, `-ep`, and `-es` Options)

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:

<code>-ea[.]new extension</code>	for an assembly language file
<code>-ec[.]new extension</code>	for a C source file
<code>-eo[.]new extension</code>	for an object file
<code>-ep[.]new extension</code>	for a C++ source file
<code>-es[.]new extension</code>	for an assembly listing file

An extension can be up to nine characters in length.

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

```
cl470 -ea=.rrr -eo .o fit.rrr
```

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

```
cl470 -ea=rrr -eoo fit.rrr
```

### 2.3.6 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:

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

```
cl470 -fb=d:\object ...
```

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

```
cl470 -ff=d:\object ...
```

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

```
cl470 -fr=d:\object ...
```

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

```
cl470 -fs=d:\assembly ...
```

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

```
cl470 -ft=d:\temp ...
```

### 2.3.7 Options That Control the Assembler

These are the assembler options that you can use with the compiler:

<b>-aa</b>	Invokes the assembler with the <code>-a</code> assembler option, which creates an absolute listing. An absolute listing shows the absolute addresses of the object code.
<b>-ac</b>	Makes case insignificant in the assembly language source files. For example, <code>-c</code> makes the symbols <code>ABC</code> and <code>abc</code> equivalent. If you do not use this option, case is significant (this is the default).
<b>-ad=name</b>	<b>-adname</b> [=value] sets the <i>name</i> symbol. This is equivalent to inserting <i>name</i> <code>.set</code> [value] at the beginning of the assembly file. If <i>value</i> is omitted, the symbol is set to 1.
<b>-ahc=filename</b>	Invokes the assembler with the <code>-hc</code> option, which 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.
<b>-ahi=filename</b>	Invokes the assembler with the <code>-hi</code> option, which 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.
<b>-al</b>	(lowercase L) Invokes the assembler with the <code>-l</code> assembler option to produce an assembly listing file.
<b>-apd</b>	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 <code>.ppa</code> extension.
<b>-api</b>	Performs preprocessing for assembly files, but instead of writing preprocessed output, writes a list of files included with the <code>#include</code> directive. The list is written to a file with the same name as the source file but with a <code>.ppa</code> extension.
<b>-as</b>	Invokes the assembler with the <code>-s</code> assembler option to put labels in the symbol table. Label definitions are written to the COFF symbol table for use with symbolic debugging.
<b>-au=name</b>	Undefines the predefined constant <i>name</i> , which overrides any <code>-ad</code> options for the specified constant.
<b>-ax</b>	Causes the assembler to produce a symbolic cross-reference in the listing file.



For more information about assembler options, see the *TMS470R1x Assembly Language Tools User's Guide*.

### 2.3.8 Deprecated Options

Several compiler options have been deprecated. The compiler continues to accept these options, but they are not recommended for use. Future releases of the tools will not support these options. Table 2–2 lists the deprecated options and the options that have replaced them.

*Table 2–2. Compiler Backwards-Compatibility Options Summary*

Old Option	Effect	New Option
-gp	Allows function-level profiling of optimized code	-g
-gt	Enables symbolic debugging using the alternate STABS debugging format	--symdebug:coff
-gw	Enables symbolic debugging using the DWARF debugging format	--symdebug:dwarf or -g

Additionally, the `--symdebug:profile_coff` option has been added to enable function-level profiling of optimized code with symbolic debugging using the STABS debugging format (the `--symdebug:coff` or `-gt` option).

## 2.4 Using Environment Variables

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

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

### 2.4.1 Specifying Directories (C\_DIR)

The compiler uses the C\_DIR environment variable to name alternate directories that contain #include files. To specify directories for #include files, set C\_DIR with one of these command syntaxes:

Operating System	Enter
Windows™	<b>set C_DIR=</b> <i>directory1</i> [; <i>directory2</i> ...]
UNIX	<b>C_DIR="</b> <i>directory1</i> [ <i>directory2</i> ...] <b>"; export C_DIR</b>

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

### 2.4.2 Setting Default Compiler Options (C\_OPTION)

You might find it useful to set the compiler, assembler, and linker default options using the 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 the C\_OPTION 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 C\_OPTION environment variable and then reads and processes it.

The table below shows how to set C\_OPTION the environment variables. Select the command for your operating system:

Operating System	Enter
UNIX (Bourne shell)	<b>C_OPTION="</b> <i>option<sub>1</sub></i> [ <i>option<sub>2</sub></i> . . .] <b>"; export C_OPTION</b>
Windows	<b>set C_OPTION=</b> <i>option<sub>1</sub></i> [; <i>option<sub>2</sub></i> . . .]

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

```
set C_OPTION=-q -s -z
```

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

```
cl470 *.c                ; compiles and links
cl470 -c *.c              ; only compiles
cl470 *.c -z lnk.cmd      ; compiles/links using .cmd file
cl470 -c *.c -z lnk.cmd ; only compiles (-c overrides -z)
```

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 linker options, see section 4.2, *Linker Options*, on page 4-5.

## 2.5 Controlling the Preprocessor

This section describes specific features of the preprocessor. The TMS470 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 2–3.

*Table 2–3. Predefined Macro Names*

Macro Name	Description
__TMS470__	Always defined
__unsigned_chars__	Defined if char types are unsigned by default (default)
__signed_chars__	Defined if char types are signed by default
__big_endian__	Defined if parsing for big-endian code (default)
__little_endian__	Defined if parsing for little-endian code
__16bis__	Defined if parsing for 16-BIS state
__32bis__	Defined if parsing for 32-BIS state (default)
__DATE__ <sup>†</sup>	Expands to the compilation date in the form <i>mm dd yyyy</i>
__FILE__ <sup>†</sup>	Expands to the current source filename
__LINE__ <sup>†</sup>	Expands to the current line number
__TIME__ <sup>†</sup>	Expands to the compilation time in the form <i>hh:mm:ss</i>
__TI_COMPILER_VERSION__	Expands to an integer value representing the current compiler version number. For example, version 1.20 is represented as 120.
__INLINE	Expands to 1 if optimization is used; undefined otherwise. Regardless of any optimization, always undefined when <i>-pi</i> is used.

---

<sup>†</sup> Specified by the ANSI/ISO standard

You can use the names listed in Table 2–3 in the same manner as any other defined name. For example,

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

translates to a line such as:

```
printf ("%s %s" , "13:58:17", "Jan 14 1999");
```

## 2.5.2 The Search Path for #include Files

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

☐ If you enclose the filename in double quotes (" "), the compiler searches for the file in the following directories in this order:

- 1) The directory that contains the current source file. The current source file refers to the file that is being compiled when the compiler encounters the #include directive.
- 2) Directories named with the `-I` option
- 3) Directories set with the `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 `-I` option
- 2) Directories set with the `C_DIR` environment variable

See section 2.5.2.1, *Changing the #include File Search Path With the -I Option*, for information on using the `-I` option. See section 2.4.1, *Specifying Directories (C\_DIR)*, on page 2-24, for information on using the `C_DIR` environment variable.

### 2.5.2.1 Changing the #include File Search Path With the -I Option

The `-I` option names an alternate directory that contains #include files. The format of the `-I` option is:

```
-I directory1 [-I directory2 ...]
```

Each `-I` option names one *directory*. In C/C++ source, you can use the #include directive without specifying any directory information for the file; instead, you can specify the directory information with the `-I` option. For example, assume that a file called `source.c` is in the current directory. The file `source.c` contains the following directive statement:

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

Assume that the complete pathname for alt.h is:

Windows     c:\470tools\files\alt.h

UNIX         /470tools/files/alt.h

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

Operating System	Enter
Windows	cl470 -Ic:\470tools\files source.c
UNIX	cl470 -I/470tools/files source.c

### 2.5.3 Generating a Preprocessed Listing File (`-ppo` Option)

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

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

### 2.5.4 Continuing Compilation After Preprocessing (`-ppa` Option)

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

### 2.5.5 Generating a Preprocessed Listing File With Comments (`-ppc` Option)

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

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

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

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

The `-ppd` option performs preprocessing only, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility. The list is written to a file with the same name as the source file but with a `.pp` extension. Optionally, you can specify a filename for the output, for example:

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

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

The `-ppi` option performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the `#include` directive. The list is written to a file with the same name as the source file but with a `.pp` extension. Optionally, you can specify a filename for the output, for example:

```
cl55 -ppi=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

Diagnostics messages have an associated severity, as follows:

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

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

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

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:

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

- pdv** Provides verbose diagnostics that display the original source with line-wrap and indicate the position of the error in the source line
- pdw** Suppresses warning diagnostics (errors are still issued)

## 2.6.2 How You Can Use Diagnostic Suppression Options

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

Consider the following code segment:

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

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

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

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

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

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

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

### 2.6.3 Other Messages

Other error messages that are unrelated to the source, such as incorrect command-line syntax or inability to find specified files, are usually fatal. They are identified by one of the strings ">> WARNING:" or ">> ERROR:" preceding the message.

For example:

```
cl470 -j
>> WARNING: invalid option -j (ignored)
>> ERROR: no source files
```

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

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

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

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

Each source line in the raw listing file begins with one of the identifiers listed in Table 2–4.

*Table 2–4. 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 <i>#if</i> clause)
L	Change in source position, given in the following format:  L <i>line number filename key</i>  Where <i>line number</i> is the line number in the source file. The <i>key</i> is present only when the change is due to entry/exit of an include file. Possible values of <i>key</i> are as follows:  1 = entry into an include file 2 = exit from an include file

The *-pl* option also includes diagnostic identifiers as defined in Table 2–5.

*Table 2–5. 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 2–5 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 2-30.

## 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 `-O3` option, 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 (oysize Option)*, on page 3-13.

---

**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 `-pi` Compiler Option, and `-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 `-ps` 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 `-pi` option.

---

**Note: Using the `-pi` Option With `-O3` Optimizations**

When you use the `-pi` option with `-O3` 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 `-s` option. To compile and run the interlist utility on a program called `function.c`, enter:

```
cl1470 -s function.c
```

The `-s` 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 the 32-bit state. For more information about using interlist with the optimizer, see section 3.7, *Using Interlist With the Optimizer*, on page 3-14.

### *Example 2–1. An Interlisted Assembly Language File*

```
_main:
    STMFD    SP!, {LR}
;-----
; 5 | printf("Hello, world\n");
;-----
    ADR      A1, SL1
    BL       _printf
;-----
; 6 | return 0;
;-----
    MOV      A1, #0
    LDMFD    SP!, {PC}
```

# Optimizing Your Code

---

---

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

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

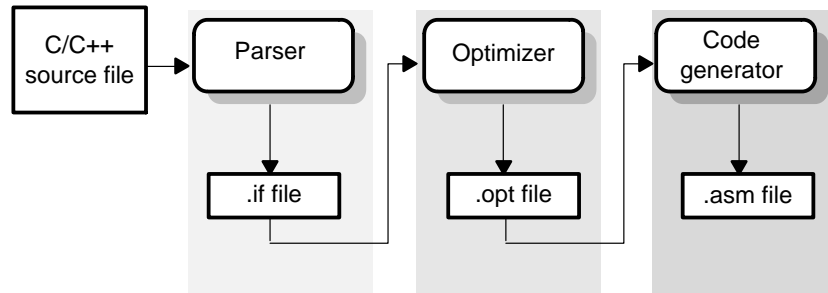
Topic	Page
3.1 Invoking Optimization .....	3-2
3.2 Performing File-Level Optimization (–O3 Option) .....	3-4
3.3 Performing Program-Level Optimization (–pm and –O3 Options) .....	3-6
3.4 Using Caution With asm Statements in Optimized Code .....	3-11
3.5 Accessing Aliased Variables in Optimized Code .....	3-12
3.6 Automatic Inline Expansion (–oi Option) .....	3-13
3.7 Using Interlist With Optimization .....	3-14
3.8 Debugging Optimized Code .....	3-16
3.9 What Kind of Optimization Is Being Performed? .....	3-18

### 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 cl470 compiler program, specifying the `-On` option on the cl470 command line. The *n* denotes the level of optimization (0, 1, 2, and 3), which controls the type and degree of optimization.

#### ☐ **-O0**

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

#### ☐ **-O1**

Performs all `-O0` optimizations, plus:

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

## ❑ **-O2**

Performs all -O1 optimizations, plus:

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

The optimizer uses -O2 as the default if you use -O without an optimization level.

## ❑ **-O3**

Performs all -O2 optimizations, plus:

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

If you use -O3, see section 3.2, *Performing File-Level Optimization (-O3 Option)*, on page 3-4 and section 3.3, *Performing Program-Level Optimization (-pm and -O3 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 (–O3 Option)

The –O3 option instructs the compiler to perform file-level optimization. You can use the –O3 option alone to perform general file-level optimization, or you can combine it with other options to perform more specific optimization. The options listed in Table 3–1 work with –O3 to perform the indicated optimization:

Table 3–1. Options That You Can Use With –O3

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

#### 3.2.1 Controlling File-Level Optimizations (–oln Option)

When you invoke the optimizer with the –O3 option, some of the optimizations use known properties of the standard library functions. If your file redeclares any of these standard library functions, these optimizations become ineffective. The –ol option (lowercase L) controls file-level optimizations. The number following –ol denotes the level (0, 1, or 2). Use Table 3–2 to select the appropriate level to append to the –ol option.

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

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

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

When you invoke the optimizer with the –O3 option, you can use the –on option to create an optimization information file that you can read. The number following the –on denotes the level (0, 1, or 2). The resulting file has a .nfo extension. Use Table 3–3 to select the appropriate level to append to the –on option.

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

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

### 3.3 Performing Program-Level Optimization (`-pm` and `-O3` Options)

You can specify program-level optimization by using the `-pm` option with the `-O3` option. With program-level optimization, all of your source files are compiled into one intermediate file called a *module*. The module moves to the optimization and code generation passes of the compiler. Because the compiler can see the entire program, it performs several optimizations that are rarely applied during file-level optimization:

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

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

In Code Composer Studio, when the `-pm` 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 `-ma`.

---

**Note: Compiling Files With the `-pm` and `-k` Options**

If you compile all files with the `-pm` and `-k` options, the compiler produces only one `.asm` file, not one for each corresponding source file.

---



### 3.3.1 Controlling Program-Level Optimization (`-opn` Option)

You can control program-level optimization, which you invoke with `-pm -O3`, by using the `-op` option. Specifically, the `-op` option indicates if functions in other modules can call a module's external functions or modify a module's external variables. The number following `-op` indicates the level you set for the module that you are allowing to be called or modified. The `-O3` option combines this information with its own file-level analysis to decide whether to treat this module's external function and variable declarations as if they had been declared static. Use Table 3–4 to select the appropriate level to append to the `-op` option.

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

If your module ...	Use this option
Has functions that are called from other modules and global variables that are modified in other modules	<code>-op0</code>
Does not have functions that are called by other modules but has global variables that are modified in other modules	<code>-op1</code>
Does not have functions that are called by other modules or global variables that are modified in other modules	<code>-op2</code>
Has functions that are called from other modules but does not have global variables that are modified in other modules	<code>-op3</code>

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

Table 3-5. *Special Considerations When Using the `-op` Option*

If your <code>-op</code> is...	Under these conditions...	Then the <code>-op</code> level...
Not specified	The <code>-O3</code> optimization level was specified	Defaults to <code>-op2</code>
Not specified	The compiler sees calls to outside functions under the <code>-O3</code> optimization level	Reverts to <code>-op0</code>
Not specified	Main is not defined	Reverts to <code>-op0</code>
<code>-op1</code> or <code>-op2</code>	No function has main defined as an entry point and functions are not identified by the <code>FUNC_EXT_CALLED</code> pragma	Reverts to <code>-op0</code>
<code>-op1</code> or <code>-op2</code>	No interrupt function is defined	Reverts to <code>-op0</code>
<code>-op1</code> or <code>-op2</code>	Functions are identified by the <code>FUNC_EXT_CALLED</code> pragma	Remains <code>-op1</code> or <code>-op2</code>
<code>-op3</code>	Any condition	Remains <code>-op3</code>

In some situations when you use `-pm` and `-O3`, you *must* use an `-op` option or the `FUNC_EXT_CALLED` pragma. See section 3.3.2, *Optimization Considerations When Mixing C and Assembly*, on page 3-8 for information about these situations.

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

If you have any assembly functions in your program, you need to exercise caution when using the `-pm` option. The compiler recognizes only the C/C++ source code and not any assembly code that might be present. Because the compiler does not recognize the assembly code calls and variable modifications to C/C++ functions, the `-pm` option optimizes out those C/C++ functions. To keep these functions, place the `FUNC_EXT_CALLED` pragma (see section 5.7.3, *The `FUNC_EXT_CALLED` Pragma*, on page 5-16) 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 `-opn` option with the `-pm` and `-O3` 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 `-pm -O3` and `-op1` or `-op2`.

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

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

**Solution** Compile with `-pm -O3 -op2` to tell the compiler that outside functions do not call C/C++ functions or modify C/C++ variables. See section 3.3.1 for information about the `-op2` option.

If you compile with the `-pm -O3` options only, the compiler reverts from the default optimization level (`-op2`) to `-op0`. The compiler uses `-op0`, because it presumes that the calls to the assembly language functions that have a definition in C/C++ may call other C/C++ functions or modify C/C++ variables.

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

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

- Compile with `-pm -O3 -op1`.
- Add the `volatile` keyword to those variables that may be modified by the assembly functions and compile with `-pm -O3 -op2`.

See section 3.3.1 on page 3-7 for information about the `-opn` 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 `-pm -O3 -op2`. *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 `-pm -O3 -op3`. Because you do not use the `FUNC_EXT_CALL` pragma, you must use the `-op3` option, which is less aggressive than the `-op2` option, and your optimization may not be as effective.

Keep in mind that if you use `-pm -O3` without additional options, the compiler removes the C functions that the assembly functions call. Use the `FUNC_EXT_CALLED` pragma to keep these functions.

### **3.4 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 `-ma` shell 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 (*-oi* Option)

When optimizing with the *-O3* option, the compiler automatically inlines small functions. The *-oysize* option specifies the size threshold. Any function larger than the *size* threshold is not automatically inlined. You can use the *-oysize* option in the following ways:

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

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

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

---

**Note: *-O3* Optimization and Inlining**

In order to turn on automatic inlining, you must use the *-O3* option. The *-O3* option turns on other optimizations. If you desire the *-O3* optimizations, but not automatic inlining, use *-oi0* with the *-O3* option.

---

---

**Note: Inlining and Code Size**

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

---

### 3.7 Using Interlist With Optimization

You control the output of the interlist feature when compiling with optimization (the `-On` option) with the `-os` and `-ss` options.

- ☐ The `-os` option interlists compiler comments with assembly source statements.
- ☐ The `-ss` and `-os` options together interlist the compiler comments and the original C/C++ source with the assembly code.

When you use the `-os` 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 `-ss` 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 `-os` option, the optimizer writes reconstructed C/C++ statements. These statements may not reflect the exact C/C++ syntax of the operation being performed.

Example 3–1 shows the function from Example 2–1 on page 2-40 compiled with optimization (`-O2`) and the `-os` option. The assembly file contains compiler comments interlisted with assembly code.

*Example 3–1. Listing for Compilation With the `-o2` and `-s` Options*

```

_main:
    STMFD    SP!, {LR}
;** 5 ----- printf("Hello, world\n");
    ADR      A1, SL1
    BL       _printf
;** 6 ----- return 0;
    MOV      A1, #0
    LDMFD    SP!, {PC}

```

When you use the `-ss` and `-os` 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 (`-o2`) and the `-ss` and `-os` options. The assembly file contains optimizer comments and C source interlisted with assembly code.



*Example 3–2. Listing for Compilation With the –o2, –os, and –ss Options*

```

_main:
    STMFD    SP!, {LR}
; ** 5 ----- printf("Hello, world\n");
;-----
; 5 | printf("Hello, world\n");
;-----
    ADR      A1, SL1
    BL      _printf
; ** 6 ----- return 0;
;-----
; 6 | return 0;
;-----
    MOV      A1, #0
    LDMFD    SP!, {PC}

```

**Note: Impact on Performance and Code Size**

The –ss 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 or the `--symdebug:coff` option (STABS debug) 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.

### 3.8.1 Debugging Optimized Code (`-g`, `--symdebug:dwarf`, `--symdebug:coff`, and `-O` Options)

To debug optimized code, use the `-O` option in conjunction with one of the symbolic debugging options (`--symdebug:dwarf` or `--symdebug:coff`). The symbolic debugging options generate directives that are used by the C/C++ source-level debugger, but they disable many compiler optimizations. When you use the `-O` option (which invokes optimization) with the `--symdebug:dwarf` or `--symdebug:coff` option, you turn on the maximum amount of optimization that is compatible with debugging.

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

---

**Note: Symbolic Debugging Options Affect Performance and Code Size**

Using the `--symdebug:dwarf` or `--symdebug:coff` option can cause a significant performance and code size degradation of your code. Use these options for debugging only. Using `--symdebug:dwarf` or `--symdebug:coff` when profiling is not recommended.

---

### 3.8.2 Profiling Optimized Code

To profile optimized code, use optimization (`-O0` through `-O3`) without debugging. Not using symbolic debugging allows you to profile optimized code at the granularity of functions.

If you have a breakpoint-based profiler, use the `--profile:breakpt` option with the `-O` option. The `--profile:breakpt` option disables optimizations that would cause incorrect behavior when using a breakpoint-based profiler.

If you have a power profiler, use the `--profile:power` option with the `-O` option. The `--profile:power` option produces instrument code for the power profiler.

If you need to profile code at a finer grain than the function level in Code Composer Studio, you can use the `--symdebug:dwarf`, or `--symdebug:coff` option, although this is not recommended. You might see a significant performance degradation because the compiler cannot use all optimizations with debugging. It is recommended that outside of Code Composer Studio, you use the `clock( )` function.

---

**Note: Profile Points**

In Code Composer Studio, when symbolic debugging is not used, profile points can only be set at the beginning and end of functions.

---

### 3.9 What Kind of Optimization Is Being Performed?

The TMS470R1x C/C++ compiler uses a variety of optimization techniques to improve the execution speed of your C/C++ programs and to reduce their size. Optimization occurs at various levels throughout the compiler.

Most of the optimizations described here are performed by the separate optimizer pass that you enable and control with the `-O` compiler options (see section 3.1, *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	3-19
Alias disambiguation	3-19
Branch optimizations and control-flow simplification	3-19
Data flow optimizations	3-21
<input type="checkbox"/> Copy propagation	
<input type="checkbox"/> Common subexpression elimination	
<input type="checkbox"/> Redundant assignment elimination	
Expression simplification	3-21
Inline expansion of runtime-support library functions	3-23
Induction variable optimizations and strength reduction	3-24
Loop-invariant code motion	3-24
Loop rotation	3-24
Tail merging	3-24
Autoincrement addressing	3-26
Block conditionalizing	3-27
Epilog inlining	3-28
Removing comparisons to zero	3-29
Integer division with constant divisor	

### 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

```
;  opt470 -O3 -e -Z fsm.if fsm.opt
;*****
;* FUNCTION DEF: _fsm                                     *
;*****
_fsm:
L2:
        LDR        A2, [A1], #4
        CMP        A2, #0
        BNE        L4
        LDR        A2, [A1], #4
        CMP        A2, #0
        BNE        L2
L4:
        LDR        A2, [A1], #4
        CMP        A2, #0
        BEQ        L4
        BX         LR
```

### 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 3-20 and Example 3–4 on page 3-22.

#### ☐ Common subexpression elimination

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

```
;*****
;* FUNCTION DEF: _simp                                     *
;*****
_simp:
    STMFD    SP!, {LR}
    ADD      A2, A1, A1, LSL #2
    MOV      A3, A1, LSL #3
    MOV      V9, A1, ASR #3
    ADD      A4, V9, A1, LSL A2
    MOV      A1, #3
    BL       _call
    LDMFD    SP!, {PC}
```





### **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. See Example 3–5 on page 3-23 and Example 3–7 on page 3-26.

### **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. See Example 3–7 on page 3-26.

### **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 3–6, 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 3–6. 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;
}

```

**(b) C/C++ compiler output**

```

;*****
;* FUNCTION DEF: _main                                     *
;*****
_main:
        STMFD     SP!, {V1, LR}
        MOVS      V1, A1
        BMI       L5
        CMP       V1, #0
        BNE       L4
        BL        _g
        MOV       V1, A1
        B         L6
L4:      MOV       A1, V1
        BL        _f
        ADD       V1, A1, V1
        B         L7
L5:      RSB       V1, V1, #0
        MOV       A1, V1
L6:      BL        _f
        ADD       V9, A1, A1, LSL #1
        ADD       V1, V9, V1
L7:      MOV       A1, V1
        LDMFD     SP!, {V1, PC}

```

### 3.9.11 Autoincrement Addressing

For pointer expressions of the form `*p++`, the compiler uses efficient TMS470 autoincrement addressing modes. In many cases, where code steps through an array in a loop such as:

```
for (i = 0; i < N; ++i) a[i]...
```

the loop optimizations convert the array references to indirect references through autoincremented register variable pointers. See Example 3–7.

#### *Example 3–7. Autoincrement Addressing, Loop-Invariant Code Motion, and Strength Reduction*

(a) C source

```
int a[10], b[10];
scale(int k)
{
    int i;
    for (i = 0; i < 10; ++i)
        a[i] = b[i] * k;
    . . .
```

(b) C/C++ compiler output

```
;  opt470 -O3 -e -Z scale.if scale.opt
;*****
;* FUNCTION DEF: _scale
;******
_scale:
        MOV     A3, #10
        LDR     A2, CON1 ; {$$+0}
L2:     LDR     V9, [A2, #40]
        MUL     V9, A1, V9
        STR     V9, [A2], #4
        SUBS    A3, A3, #1
        BNE     L2
        BX      LR
```

### 3.9.12 Block Conditionalizing

Because all 32-bit instructions can be conditional, branches can be removed by conditionalizing instructions.

In Example 3–8, the branch around the add and the branch around the subtract are removed by simply conditionalizing the add and the subtract.

#### Example 3–8. Block Conditionalizing

(a) C source

```
int main(int a)
{
    if (a < 0)
        a = a-3;
    else
        a = a*3;

    return ++a;
}
```

(b) C/C++ compiler output

```
;*****
;* FUNCTION DEF: _main
;* *****
_main:
        CMP        A1, #0
        ADDPL      A1, A1, A1, LSL #1
        SUBMI      A1, A1, #3
        ADD        A1, A1, #1
        BX         LR
```

### 3.9.13 Epilog Inlining

If the epilog of a function is a single instruction, that instruction replaces all branches to the epilog. This increases execution speed by removing the branch.

In Example 3–9, because the epilog is a single instruction, the branch to the epilog after the multiplication is replaced with the epilog itself.

#### *Example 3–9. Epilog Inlining*

(a) C source

```
int main(int a)
{
    if (a < 0)
        a = a-3;
    else
        a = a*3;

    return a;
}
```

(b) C/C++ compiler output

```
;*****
;* FUNCTION DEF: $main
;* *****
$main:
        CMP        A1, #0
        BMI        L3
        LSL        A2, A1, #1
        ADD        A1, A1, A2
        BX         LR
L3:
        SUB        A1, #3
        BX         LR
```

### 3.9.14 Removing Comparisons to Zero

Because most of the 32-bit instructions and some of the 16-bit instructions can modify the status register, explicit comparisons to 0 may be unnecessary. The TMS470 C/C++ compiler removes comparisons to 0 if a previous instruction can be modified to set the status register appropriately.

In Example 3–10, the explicit comparison to 0 following the SUB instruction was removed, and SUB was modified to set the status register itself.

#### Example 3–10. Removing Comparisons to Zero

(a) C code

```
int main(int a)
{
    a -= 1;

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

(b) C/C++ compiler output

```
;*****
;* FUNCTION DEF: _main
;* *****
_main:
        SUBS      A1, A1, #1
        RSBLT     A1, A1, #0
        BX        LR
```

### 3.9.15 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 -O2 and higher. You must also compile with the -mf options, which selects compile for speed.

---



# Linking C/C++ Code

The TMS470R1x 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 linker 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 linker, see the *TMS470R1x Assembly Language Tools User's Guide*.

Topic	Page
4.1 Invoking the Linker (-z Option) .....	4-2
4.2 Linker Options .....	4-5
4.3 Controlling the Linking Process .....	4-7

## 4.1 Invoking the Linker (-z Option)

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

### 4.1.1 Invoking the Linker As a Separate Step

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

```
cl470 -z {-c|-cr} filenames [options] [-o name.out] [lnk.cmd] -l library
```

<b>cl470 -z</b>	The command that invokes the linker
<b>-c   -cr</b>	Options that tell the linker to use special conventions defined by the C/C++ environment. When you use <code>cl470 -z</code> , you must use <code>-c</code> or <code>-cr</code> . The <code>-c</code> option uses automatic variable initialization at run time; the <code>-cr</code> option uses automatic variable initialization at load time.
<i>options</i>	Options affect how the linker handles your object files. Options can appear anywhere on the command line or in a linker command file. (Options are discussed in section 4.2, <i>Linker Options</i> .)
<i>filenames</i>	Names of object files, linker command files, or archive libraries. The default extension for all input files is <code>.obj</code> ; any other extension must be explicitly specified. The linker can determine whether the input file is an object or ASCII file that contains linker commands. The default output filename is <code>a.out</code> , unless you use the <code>-o</code> option to name the output file.
<b>-o name.out</b>	The <code>-o</code> option names the output file.
<i>lnk.cmd</i>	Contains options, filenames, directives, or commands for the linker.
<b>-l library</b>	(lowercase L) Identifies the appropriate archive library containing C/C++ run-time-support and floating-point math functions. (The <code>-l</code> option tells the linker that a file is an archive library.) You can use the libraries included with the compiler, or you can create your own run-time-support library. If you have specified a run-time-support library in a linker command file, you do not need this parameter.

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

The linker uses a default allocation algorithm to allocate your program into memory. You can use the MEMORY and SECTIONS directives in the linker command file to customize the allocation process. For more information, see the *TMS470R1x 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:

```
cl470 -z -c prog1 prog2 prog3 -o prog.out -l rtsc_32.lib
```

#### 4.1.2 Invoking the Linker as Part of the Compile Step

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

```
cl470 [options] filenames -z {-c|-cr} filenames [options] [-o name.out]
                               [lnk.cmd] -l library
```

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

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

All arguments that precede -z on the command line are compiler arguments. 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:

```
cl470 prog1.c prog2.c prog3.c -z -c -o prog.out -l rts55.lib
```

**Note: Order of Processing Arguments in the Linker**

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

- 1) Object filenames from the command line
- 2) Arguments following the -z option on the command line
- 3) Arguments following the -z option from the C\_OPTION environment variable

### 4.1.3 Disabling the Linker (-c Option)

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

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

## 4.2 Linker Options

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

<code>-a</code>	Produces an absolute, executable module. This is the default; if neither <code>-a</code> nor <code>-r</code> is specified, the linker acts as if <code>-a</code> is specified.
<code>-ar</code>	Produces a relocatable, executable object module
<code>--args=size</code>	Allocates memory to be used by the loader to pass arguments from the command line of the loader to the program. The linker allocates <i>size</i> bytes in an uninitialized <code>.args</code> section. The <code>__c_args__</code> symbol contains the address of the <code>.args</code> section.
<code>-c</code>	Autoinitializes variables at run time
<code>-cr</code>	Autoinitializes variables at load time
<code>-e global_symbol</code>	Defines a <i>global_symbol</i> that specifies the primary entry point for the output module
<code>-f fill_value</code>	Sets the default fill value for holes within output sections; <i>fill_value</i> must be a 32-bit constant
<code>-g global_symbol</code>	Defines a <i>global_symbol</i> as global even if the global symbol has been made static with the <code>-h</code> linker option
<code>-h</code>	Makes all global symbols static
<code>-heap size</code>	Sets heap size (for the dynamic memory allocation) to <i>size</i> bytes and defines a global symbol that specifies the heap size. The default is 2048 bytes.
<code>-i directory</code>	Alters the library-search algorithm to look in <i>directory</i> before looking in the default location(s). This option must appear before the <code>-l</code> linker option. The directory you specify must follow operating system conventions.
<code>-j</code>	Disables conditional linking. The C/C++ compiler automatically generates the <code>.clink</code> assembler directive with every section that is a candidate for conditional linking; the <code>-j</code> option overrides this default. For more information about conditional linking and the <code>.clink</code> assembler directive, see the <i>TMS470R1x Assembly Language Tools User's Guide</i> .

<b>-l</b> <i>filename</i>	(lowercase L) Names an archive library file as linker input and instructs the linker to use the C_DIR environment variable and paths specified with the -i option to help find the archive library; <i>filename</i> is an archive library name and must follow operating system conventions.
<b>-m</b> <i>filename</i>	Produces a map or listing of the input and output sections, including holes, and places the listing in <i>filename</i> . The filename you specify must follow operating system conventions.
<b>-o</b> <i>filename</i>	Names the executable output module. The <i>filename</i> must follow operating system conventions. If the -o option is not used, the default filename is a.out.
<b>-q</b>	Requests a quiet run (suppresses the banner)
<b>-r</b>	Retains relocation entries in the output module
<b>-s</b>	Strips symbol table information and line number entries from the output module
<b>-stack</b> <i>size</i>	Sets the C/C++ system stack size to <i>size</i> bytes and defines a global symbol that specifies the stack size. The default is 2048 bytes.
<b>-u</b> <i>symbol</i>	Places the unresolved external symbol <i>symbol</i> into the output module's symbol table
<b>-w</b>	Displays a message when an undefined output section is created
<b>-x</b>	Forces rereading of libraries to resolve backward symbol references

For more information on linker options, see the linker description chapter of the *TMS470R1x Assembly Language Tools User's Guide*.

## 4.3 Controlling the Linking Process

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

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

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

For more information about how to operate the linker, see the *Linker Description* chapter of the *TMS470R1x 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 `-l` linker option to specify the run-time-support library (`rtsc_16.lib`, or `rtsc_32.lib`) to use. The `-l` option also tells the linker to look at the `-I` options and then the `C_DIR` environment variable to find an archive path or object file.

To use the `-l` option, type on the command line:

**cl470 -z {-c | -cr} filenames -l libraryname**

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

### 4.3.2 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) Switches to user mode and sets up the user mode stack
- 2) Set up status and configuration registers
- 3) Set up the stack and secondary system stack
- 4) Process the .cinit run-time initialization table to autoinitialize global variables (when using the `-c` option)
- 5) Calls all global constructors (.pinit)
- 6) Calls main
- 7) 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 `-c` or `-cr` linker 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. A sample interrupt vector is provided in `vectors.obj` in `rts55.lib`. For C55x, the first few lines of the vector are:

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



### 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 6.9.4, *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 6.9.4, *Initialization Tables*, on page 6-37 discusses the format of these tables. These tables are in a named section called `.cinit`. The initialization tables are used in one of the following ways:

- ☐ Autoinitializing variables at run time. Global variables are initialized at *run time*. Use the `-c` linker option (see section 6.9.5, *Autoinitialization of Variables at Run Time*, on page 6-39).
- ☐ Initializing variables at load time. Global variables are initialized at *load time*. Use the `-cr` linker option (see section 6.9.6, *Initialization of Variables at Load Time*, on page 6-40).

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

- ☐ The symbol `_c_int00` is defined as the program entry point; it identifies the beginning of the C boot routine in `boot.obj`. When you use `-c` or `-cr`, `_c_int00` is automatically referenced; 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 `-cr` linker option), the following occur:
  - The linker sets the symbol `cinit` to `-1`. This indicates that the initialization tables are not in memory, so no initialization is performed at run time.
  - The `STYP_COPY` flag is set in the `.cinit` section header. `STYP_COPY` is the special attribute that tells the loader to perform autoinitialization directly and not to load the `.cinit` section into memory. The linker does not allocate space in memory for the `.cinit` section.
- ☐ When autoinitializing at run time (`-c` linker option), the linker defines the symbol `cinit` as the starting address of the `.cinit` section. The boot routine uses this symbol as the starting point for autoinitialization.
- ☐ The linker 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 C/C++ compiler tools. You can use the TMS470 simulator or emulator with the source debugger as a loader.

---

### 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 4–1 summarizes the sections.

*Table 4–1. Sections Created by the Compiler*

(a) *Initialized sections*

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

Table 4–1. Sections Created by the Compiler(Continued)

(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 linker provides `MEMORY` and `SECTIONS` directives for allocating sections. For more information about allocating sections into memory, see the linker description chapter of the *TMS470 Assembly Language Tools User's Guide*.

### 4.3.7 A Sample Linker Command File

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

<b>-c</b>	Tells the linker to use autoinitialization at run time
<b>-stack</b>	Tells the linker to set the C stack size at 0x8000 bytes
<b>-heap</b>	Tells the linker to set the heap size to 0x2000 bytes
<b>-l</b>	Tells the linker to use an archive library file, <code>rtsc_32.lib</code>

To link the program, enter:

```
cl470 -z object_file(s) -o outfile -m mapfile lnk32.cmd
```

*Example 4-1. Linker Command File*

```

/*****
/* LNK32.CMD - V1.00  COMMAND FILE FOR LINKING TMS470 (ARM) C/C++ PROGRAMS */
/*
/* Usage: lnk470 <obj files...> -o <out file> -m <map file> lnk32.cmd */
/* cl470 <src files...> -z -o <out file> -m <map file> lnk32.cmd*/
/*
/* Description: This file is a sample command file that can be used */
/* for linking programs built with the TMS470 C/C++ */
/* Compiler. Use it as a guideline; you may want to change */
/* the allocation scheme according to the size of your */
/* program and the memory layout of your target system. */
/*
/* Notes: (1) If you are building a C++ application, use the C++ */
/* runtime library rtscpp_32.lib rather than the C runtime */
/* library rtsc_32.lib. */
/*
/* (2) You must specify the directory in which rtsc_32.lib or */
/* rtscpp_32.lib is located. Either add a "-i<directory>" */
/* line to this file, or use the system environment */
/* variable C_DIR to specify a search path for libraries */
/*
/* (3) If the run-time support library you are using is not */
/* named rtsc_32.lib, or rtscpp_32.lib, be sure to use the */
/* correct name here. */
/*
/*****
-c /* LINK USING C CONVENTIONS */
-stack 0x8000 /* SOFTWARE STACK SIZE */
-heap 0x2000 /* HEAP AREA SIZE */
-l rtsc_32.lib /* GET RUN-TIME SUPPORT */

/* SPECIFY THE SYSTEM MEMORY MAP */
MEMORY
{
    P_MEM : org = 0x00000000 len = 0x00030000 /* PROGRAM MEMORY (ROM) */
    D_MEM : org = 0x00030000 len = 0x00050000 /* DATA MEMORY (RAM) */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */
SECTIONS
{
    .intvecs : {} > 0x0 /* INTERRUPT VECTORS */
    .bss : {} > D_MEM /* GLOBAL & STATIC VARS */
    .systemem : {} > D_MEM /* DYNAMIC MEMORY ALLOCATION AREA */
    .stack : {} > D_MEM /* SOFTWARE SYSTEM STACK */
    .text : {} > P_MEM /* CODE */
    .pinit : {} > P_MEM /* INITIALIZATION ROUTINE TABLES */
    .cinit : {} > P_MEM /* INITIALIZATION TABLES */
    .const : {} > P_MEM /* CONSTANT DATA */
}

```

### 4.3.8 Using Function Subsections (`-ms` Compiler Option)

When the linker 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 `-ms` 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.

---

# TMS470R1x C and C++ Languages

The TMS470R1x C/C++ compiler supports the C language standard developed by a committee of the International Organization for Standardization (ISO) to standardize the C programming language.

The C++ language supported by the TMS470 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 TMS470R1x C .....	5-2
5.2 Characteristics of TMS470R1x C++ .....	5-5
5.3 Data Types .....	5-6
5.4 Keywords .....	5-7
5.5 Register Variables .....	5-10
5.6 The asm Statement .....	5-13
5.7 Pragma Directives .....	5-14
5.8 Generating Linknames .....	5-23
5.9 Initializing Static and Global Variables .....	5-24
5.10 Changing the Language Mode (–pk, –pr, and –ps Options) .....	5-26

## 5.1 Characteristics of TMS470R1x 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 TMS470 C/C++ compiler.

The following list identifies all such cases and describes the behavior of the TMS470 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 5.5, *Register Variables*, on page 5-10. (ISO 3.5.1, K&R A2.1)
- ☐ Structure members are packed into 32-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. For more information about bit-field packing, see section 6.2.2, *Bit Fields*, page 6-11. (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:

- DATA\_SECTION
- DUAL\_STATE
- FUNC\_EXT\_CALLED
- INTERRUPT
- MUST\_ITERATE
- TASK
- SWI\_ALIAS
- UNROLL

For more information on pragmas, see section 5.7, *Pragma Directives*, on page 5-14.

## 5.2 Characteristics of TMS470R1x C++

The TMS470 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:
  - <locale>
  - <signal>
  - <wchar>
  - <wctype>
- ☐ These C++ headers are the only C++ standard library header files included:
  - <new>
  - <typeinfo>
  - <ciso646>
- ☐ Exception handling is not supported.
- ☐ 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 TMS470 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 7-19.

Table 5–1. TMS470R1x C Data Types

Type	Size	Representation	Minimum Value	Range
				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	–32768	32767
unsigned short, wchar_t	16 bits	Binary	0	65 535
int, signed int	32 bits	2s complement	–2 147 483 648	2 147 483 647
unsigned int	32 bits	Binary	0	4 294 967 295
long, signed long	32 bits	2s complement	–2 147 483 648	2 147 483 647
unsigned long	32 bits	Binary	0	4 294 967 295
long long, signed long long	64 bits	2s complement	–9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 bits	Binary	0	18 446 744 073 709 551 615
enum	32 bits	2s complement	–2 147 483 648	2 147 483 647
float	32 bits	IEEE 32-bit	1.175495e–38	3.40282346e+38
double	64 bits	IEEE 64-bit	2.22507386e–308	1.79769313e+308
long double	64 bits	IEEE 64-bit	2.22507385e–308	1.79769313e+308
pointers, references, pointer to data members	32 bits	Binary	0	0xFFFFFFFF

## 5.4 Keywords

The TMS470 C/C++ compiler supports the standard `const` and `volatile` keywords. In addition, the TMS470 C/C++ compiler extends the C language through the support of the `interrupt` keyword.

### 5.4.1 The `const` Keyword

The TMS470R1x 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 TMS470 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.

Unless specified otherwise by the `INTERRUPT` pragma, the interrupt keyword defines the function as an IRQ (interrupt request) interrupt type. For more information, see section 5.7.4, *The INTERRUPT Pragma*, on page 5-17.

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

### 5.4.3 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 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.

### 5.5.1 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 6-13.



## 5.5.2 Global Register Variables

The C/C++ compiler extends the C language by adding a special convention to the register storage class specifier to allow the allocation of global registers. This special global declaration has the form:

**register type *regid***

where *regid* can be `__R5`, `__R6`, or `__R9`

The identifiers `__R5`, `__R6`, and `__R9` are each bound to their corresponding register R5, R6 and R9, respectively.

When you use this declaration at the file level, the register is permanently reserved from any other use by the optimizer and code generator for that file. You cannot assign an initial value to the register. You can use a `#define` directive to assign a meaningful name to the register; for example:

```
register struct data_struct *__R5
#define data_pointer __R5
data_pointer->element;
data_pointer++;
```

There are two reasons that you would be likely to use a global register variable:

- ☐ You are using a global variable throughout your program, and it would significantly reduce code size and execution speed to assign this variable to a register permanently.
- ☐ You are using an interrupt service routine that is called so frequently that it would significantly reduce execution speed if the routine did not have to save and restore the register(s) it uses every time it is called.

You need to consider very carefully the implications of reserving a global register variable. Registers are a precious resource to the compiler, and using this feature indiscriminately may result in poorer code.

You also need to consider carefully how code with a globally declared register variable interacts with other code, including library functions, that does not recognize the restriction placed on the register.

Because the registers that can be global register variables are save-on-entry registers, a normal function call and return does not affect the value in the register and neither does a normal interrupt. However, when you mix code that has a globally declared register variable with code that does not have the register reserved, it is still possible for the value in the register to become corrupted. To avoid the possibility of corruption, you must follow these rules:

- ☐ Functions that alter global register variables cannot be called by functions that are not aware of the global register. Use the `-r` shell option to reserve the register in code that is not aware of the global register declaration. You must be careful if you pass a pointer to a function as an argument. If the passed function alters the global register variable and the called function saves the register, the value in the register will be corrupted.
- ☐ You cannot access a global register variable in an interrupt service routine unless you recompile all code, including all libraries, to reserve the register. This is because the interrupt routine can be called from any point in the program.
- ☐ The `longjump ()` function restores global register variables to the values they had at the `setjump ()` location. If this presents a problem in your code, you must alter the code for the function and recompile `rts.src`.

The `-r register` command-line option for the `cl470` compiler allows you to prevent the compiler from using the named *register*. This lets you reserve the named register in modules that do not have the global register variable declaration, such as the runtime-support libraries, if you need to compile the modules to prevent some of the above occurrences.

## 5.6 The asm Statement

The TMS470R1x C/C++ compiler can embed TMS470 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 *TMS470R1x 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 `-ps` shell 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.

## 5.7 Pragma Directives

Pragma directives tell the compiler's preprocessor how to treat functions. The TMS470 C/C++ compiler supports the following pragmas:

- ☐ DATA\_SECTION
- ☐ DUAL\_STATE
- ☐ FUNC\_EXT\_CALLED
- ☐ INTERRUPT
- ☐ TASK
- ☐ SWI\_ALIAS

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.

### 5.7.1 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 5–1. 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,512,4
.global _bufferB
_bufferB: .usect  "my_sect",512,4
```

## 5.7.2 The DUAL\_STATE Pragma

By default (that is, without the shell's `–md` option), all functions with external linkage support dual-state interworking. This support assumes that most calls do not require a state change and are therefore optimized (in terms of code size and execution speed) for calls not requiring a state change. Using the `DUAL_STATE` pragma does not change the functionality of the dual-state support, but it does assert that calls to the applied function often require a state change. Therefore, such support is optimized for state changes.

The syntax of the pragma in C is:

```
#pragma DUAL_STATE (func)
```

The argument *func* is the function to which the pragma is applied.

The syntax of the pragma in C++ is:

```
#pragma DUAL_STATE
```

For more information, see section 6.10, *Dual-State Interworking*, on page 6-41.

### 5.7.3 The FUNC\_EXT\_CALLED Pragma

When you use program-level optimization (file-level optimization, using the `-o3` option, 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
```

### 5.7.4 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 type of interrupt is specified by the pragma; the IRQ (interrupt request) interrupt type is assumed if none is given.

The syntax of the pragma in C is:

```
#pragma INTERRUPT (func[, interrupt_type] )
```

The syntax of the pragma in C++ is:

```
#pragma INTERRUPT [ (interrupt_type) ]
```

The argument *func* is the name of a function, and the argument *interrupt\_type* is an interrupt type. The argument *interrupt\_type* is optional. The registers that are saved and the return sequence depend upon the interrupt type. If the interrupt type is omitted from the interrupt pragma, the interrupt type IRQ is assumed. These are the valid interrupt types:

Interrupt Type	Description
DABT	Data abort
FIQ	Fast interrupt request
IRQ	Interrupt request
PABT	Prefetch abort
RESET	System reset
SWI	Software interrupt
UDEF	Undefined instruction

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.

### 5.7.5 The MUST\_ITERATE Pragma

The MUST\_ITERATE pragma specifies to the compiler certain properties of a loop. You guarantee that these properties are always true. Through the use of the MUST\_ITERATE pragma, you can guarantee that a loop executes a specific number of times. Anytime the UNROLL pragma is applied to a loop, MUST\_ITERATE should be applied to the same loop. Here the MUST\_ITERATE pragma's third argument, *multiple*, is the most important and should always be specified.

Furthermore, the MUST\_ITERATE pragma should be applied to any other loops as often as possible. This is because the information provided via the pragma (especially the minimum number of iterations) aids the compiler in choosing the best loops and loop transformations. It also helps the compiler reduce code size.

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

#### 5.7.5.1 The MUST\_ITERATE Pragma Syntax

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

```
#pragma MUST_ITERATE (min, max, multiple);
```

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

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

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

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

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



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

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

### 5.7.5.2 Using `MUST_ITERATE` to Expand Compiler Knowledge of Loops

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

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

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

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

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

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

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

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

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

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

### 5.7.6 The TASK Pragma

The TASK pragma specifies that the function to which it is applied is a task. Tasks are functions that are called but never return. Typically, they consist of an infinite loop that simply dispatches other activities. Because they never return, there is no need to save (and therefore restore) registers that would otherwise be saved and restored. This can save RAM space, as well as some code space.

The syntax of the pragma in C is:

```
#pragma TASK (func)
```

The syntax of the pragma in C++ is:

```
#pragma TASK
```

### 5.7.7 The SWI\_ALIAS Pragma

The SWI\_ALIAS pragma allows you to refer to a particular software interrupt as a function name and to invocations of the software interrupt as function calls. Since the function name is simply an alias for the software interrupt, no function definition exists for the function name.

The syntax for the SWI\_ALIAS pragma in C is:

```
#pragma SWI_ALIAS (func, swi_number)
```

The syntax for the SWI\_ALIAS pragma in C++ is:

```
#pragma SWI_ALIAS (swi_number)
```

Calls to the applied function are compiled as software interrupts whose number is *swi\_number*. The *swi\_number* variable must be an integer constant.

A function prototype must exist for the alias and it must occur after the pragma and before the alias is used. Software interrupts whose number is not known until run time are not supported.

For more information about using software interrupts, including restrictions on passing arguments and register usage, see section 6.6.5, *Using Software Interrupts*, on page 6-28.

**Example 5–2. Using the SWI\_ALIAS Pragma****(a) C source file**

```
#pragma SWI_ALIAS(put, 48);    /* #pragma SWI_ALIAS(48) for C++ */

int put (char *key, int value);
void error();

main()
{
    if (!put("one", 1)) /* calling "put" invokes SWI #48 with 2 arguments */
        error();        /* and returns a result. */
}
```

**(b) Resulting assembly source file**

```
;*****
;* FUNCTION DEF: _main
;* *****
_main:
    STMFD    SP!, {LR}
    ADR      A1, SL1
    MOV      A2, #1
    SWI      #48           ; SWI #48 is generated for the function call
    CMP      A1, #0
    BLEQ     _error
    MOV      A1, #0
    LDMFD    SP!, {PC}

SL1:      .string "one",0
```

### 5.7.8 The UNROLL Pragma

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

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

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

```
#pragma UNROLL (n);
```

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

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

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

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

Specifying `#pragma UNROLL(1);` asks that the loop not be unrolled. Automatic loop unrolling also is not performed in this case.

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

## 5.8 Generating Linknames

The compiler transforms the names of externally visible identifiers when creating their linknames. The algorithm used depends on the scope within which the identifier is declared. For objects, an underscore ( `_` ) is prepended to the identifier name. For C functions, 32-bit mode functions have an underscore ( `_` ) prepended to their names and 16-bit mode functions have a dollar sign ( `$` ) prepended. C++ functions have the same initial character ( `_` or `$` ) prepended, but also have the function name further mangled. Mangling is the process of embedding a function's signature (the number and types of its parameters) into its name. The mangling algorithm used closely follows that described in *The C++ Annotated Reference Manual* (ARM). Mangling allows function overloading, operator overloading, and type-safe linking.

The general form of a C++ linkname for a 32-bit function named *func* is:

`__func__Fparmcodes`

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 compiled in 16-bit mode
```

this is the resulting assembly:

```
$__foo__Fi;
```

The linkname of *foo* is `$__foo__Fi`, indicating that *foo* is a 16-bit 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.

## 5.9 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.

### 5.9.1 Initializing Static and Global Variables With the Linker

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

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

Because the linker writes a complete load image of the zeroed .bss section into the output COFF file, this method can have the unwanted effect of significantly increasing the size of the output file (but not the program).

If you burn your application into ROM, you should explicitly initialize variables that require initialization. The method demonstrated 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 linker command files and the SECTIONS directive, see the linker description chapter in the *TMS470R1x Assembly Language Tools User's Guide*.

## 5.9.2 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 5.9, *Initializing Static and Global Variables*). For example:

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

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

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

corresponds to an entry in the *.const* section:

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

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

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
```

## 5.10 Changing the Language Mode (*-pk*, *-pr*, and *-ps* Options)

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

- ☐ Normal ANSI/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.

### 5.10.1 Compatibility With K&R C (*-pk* Option)

The ANSI/ISO C language is basically a superset of the de facto C standard defined in Kernighan and Ritchie's *The C Programming Language*. Most programs written for other non-ANSI/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 TMS470 ANSI/ISO C compiler, the compiler has a K&R C option (*-pk*) that modifies some semantic rules of the language for compatibility with older code. In general, the *-pk* option relaxes requirements that are stricter for ANSI/ISO C than for K&R C. The *-pk* option does not disable any new features of the language such as function prototypes, enumerations, initializations, or preprocessor constructs. Instead, *-pk* simply liberalizes the ANSI/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 `-pk` is used, but with less severity:

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

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

```
a; /* illegal unless -pk 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 -pk 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 -pk 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 -pk used, error
               if not */
```

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

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

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

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

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

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

### 5.10.2 Enabling Strict ANSI/ISO Mode and Relaxed ANSI/ISO Mode (*-ps* and *-pr* Options)

Use the *-ps* 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 *-pr* 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.

### 5.10.3 Enabling Embedded C++ Mode (*-pe* Option)

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

- ☐ Templates
- ☐ Exception handling
- ☐ 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 TMS470 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.

## 5.11 Compiler Limits

Due to the variety of host systems supported by the TMS470 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 shell continues compiling the file with the code generator. This results in a file compiled with no optimization. The optimizer compiles one function at a time, so the most likely cause of this is a large or extremely complex function in your source module. To correct the problem, your options are:

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

---

# Run-Time Environment

This chapter describes the TMS470 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 .....	6-2
6.2 Object Representation .....	6-6
6.3 Register Conventions .....	6-13
6.4 Function Structure and Calling Conventions .....	6-15
6.5 Interfacing C/C++ With Assembly Language .....	6-20
6.6 Interrupt Handling .....	6-26
6.7 Intrinsic Run-Time-Support Arithmetic and Conversion Routines .....	6-30
6.8 Built-In Functions .....	6-34
6.9 System Initialization .....	6-35
6.10 Dual-State Interworking .....	6-41

## 6.1 Memory Model

The TMS470 treats memory as a single linear 32-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 linker.

---

**Note: The Linker Defines the Memory Map**

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

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

---

### 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 *TMS470R1x 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 5.7.1, *The DATA\_SECTION Pragma*, on page 5-14 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 linker 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
<i>.bss</i>	RAM	<i>.pinit</i>	ROM or RAM
<i>.cinit</i>	ROM or RAM	<i>.stack</i>	RAM
<i>.const</i>	ROM or RAM	<i>.sysmem</i>	RAM
<i>.data</i>	ROM or RAM	<i>.text</i>	ROM or RAM

You can use the *SECTIONS* directive in the linker command file to customize the section-allocation process. For more information about allocating sections into memory, see the linker description chapter in the *TMS470R1x 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 R13 register to manage the stack. R13 is the *stack pointer* (SP), which points to the top of the stack (the highest address used).

The linker 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 2K bytes. You can change the size of the stack at link time by using the `-stack` option with the linker command. For more information on the stack option, see section 4.2, *Linker Options*, on page 4-5.

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 6-13; for more information about the stack, see section 6.4, *Function Structure and Calling Conventions*, on page 6-15.)

---

**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 `.system` section. You can set the size of the `.system` section by using the `-heap size` option with the linker command. The linker also creates a global symbol, `__SYSTEM_SIZE`, and assigns it as a value equal to the size of the heap in bytes. The default size is 2K bytes. For more information on the `-heap` option, see section 4.2, *Linker 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 (`.system`); 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 linker by using the `-cr` linker option. For more information, see section 6.9, *System Initialization*, on page 6-35.

## 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 <sup>†</sup>	8 bits
unsigned char, bool	Bits 0–7 of register	8 bits
short, signed short	Bits 0–15 of register <sup>†</sup>	16 bits, halfword aligned
unsigned short, wchar_t	Bits 0–15 of register	16 bits, halfword aligned
int, signed int	Bits 0–31 of register	32 bits, word (32 bits) aligned
unsigned int	Bits 0–31 of register	32 bits, word (32 bits) aligned
enum	Bits 0–31 of register	32 bits, word (32 bits) aligned
long, signed long	Bits 0–31 of register	32 bits, word (32 bits) aligned
unsigned long	Bits 0–31 of register	32 bits, word (32 bits) aligned
long long	Even/odd register pair	64 bits, word (32 bits) aligned
unsigned long long	Even/odd register pair	64 bits, word (32 bits) aligned
double	Even/odd register pair	64 bits aligned to 64-bit boundary
float	Bits 0–31 of register	32 bits, word (32 bits) aligned
double	Register pair	64 bits, word (32 bits) aligned
long double	Register pair	64 bits, word (32 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 0–31 of register	32 bits, word (32 bits) aligned
pointer to member function	Members stored as their individual types require	64 bits, word (32 bits) aligned

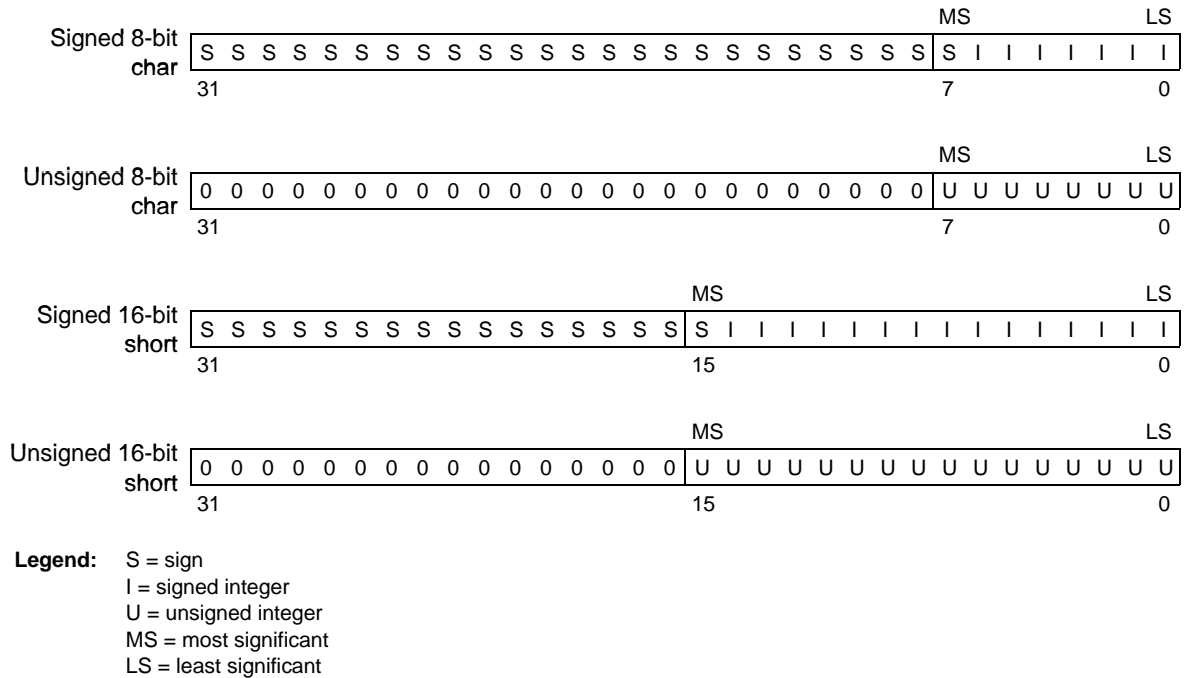
---

<sup>†</sup> Negative values are sign-extended to bit 31.

### 6.2.1.1 Char, Unsigned Char, Short, and Unsigned Short Data Types

Char and unsigned char objects are stored in memory as a single byte, and they are loaded to and stored from bits 0–7 of a register (see Figure 6–1). Objects defined as short or unsigned short are stored in memory as two bytes at a halfword (2 byte) aligned address, and they are loaded to and stored from bits 0–15 of a register (see Figure 6–1).

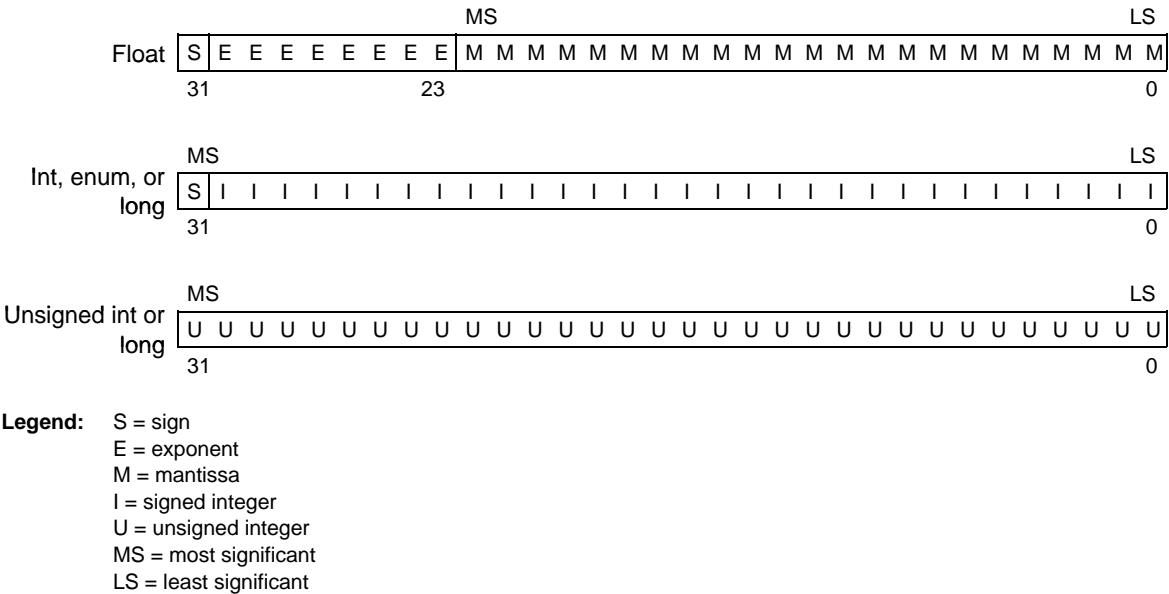
Figure 6–1. Char and Short Data Storage Format



6.2.1.2 Int, Unsigned Int, Enum, Float, Long, and Unsigned Long Data Types

The int, unsigned int, enum, float, long, and unsigned long data types are stored in memory as 32-bit objects at word (4-byte) aligned addresses. Objects of these types are loaded to and stored from bits 0–32 of a register, as shown in Figure 6–2. In big-endian mode, 4-byte objects are loaded to registers by moving the first byte (the lower address) of memory to bits 24–31 of the register, moving the second byte of memory to bits 16–23, moving the third byte to bits 8–15, and moving the fourth byte to bits 0–7. In little-endian mode, 4-byte objects are loaded to registers by moving the first byte (the lower address) of memory to bits 0–7 of the register, moving the second byte to bits 8–15, moving the third byte to bits 16–23, and moving the fourth byte to bits 24–31.

Figure 6–2. 32-Bit Data Storage Format



### 6.2.1.3 *Pointer to Data Member Data Type*

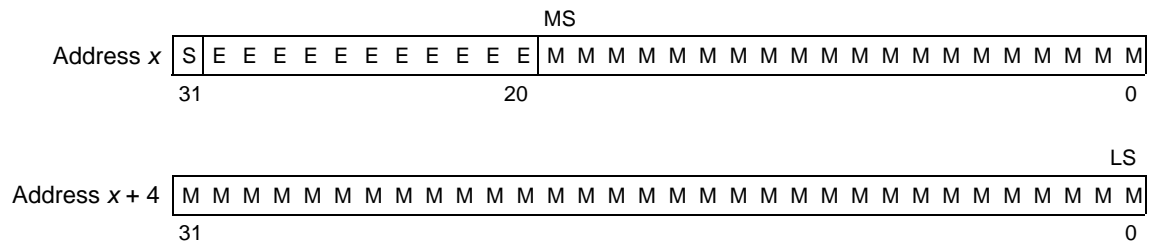
Pointer to data member objects are stored in memory like an unsigned int data type, as seen in section 6.2.1.2. Its value is the byte offset to the data member in the class plus 1, so that the zero value is reserved to represent the null pointer to data member.

#### 6.2.1.4 Double, Long Double, and Long Long Data Types

Double, long double, and long long data types are stored in memory in a pair of registers and are always referenced as a pair. These types are stored as 64-bit objects at word (4-byte) aligned addresses. The word at the lowest address contains the sign bit, the exponent, and the most significant part of the mantissa. The word at the higher address contains the least significant part of the mantissa. This is true regardless of the endianness of the target. However, the ordering of bytes within each word depends on the endianness of the target.

Objects of this type are loaded into and stored in register pairs, as shown in Figure 6–3. The most significant memory word contains the sign bit, exponent, and the most significant part of the mantissa. The least significant memory word contains the least significant part of the mantissa.

Figure 6-3. Double-Precision Floating-Point Data in Register Pairs



**Legend:** S = sign  
E = exponent  
M = mantissa  
MS = most significant  
LS = least significant

### 6.2.1.5 *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.

### 6.2.1.6 *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.

## 6.2.2 Bit Fields

Bit fields are the only objects that can be packed within a byte. That is, two bit fields can be stored in the same byte. Bit fields can range in size from 1 to 32 bits, but they never span a 4-byte boundary.

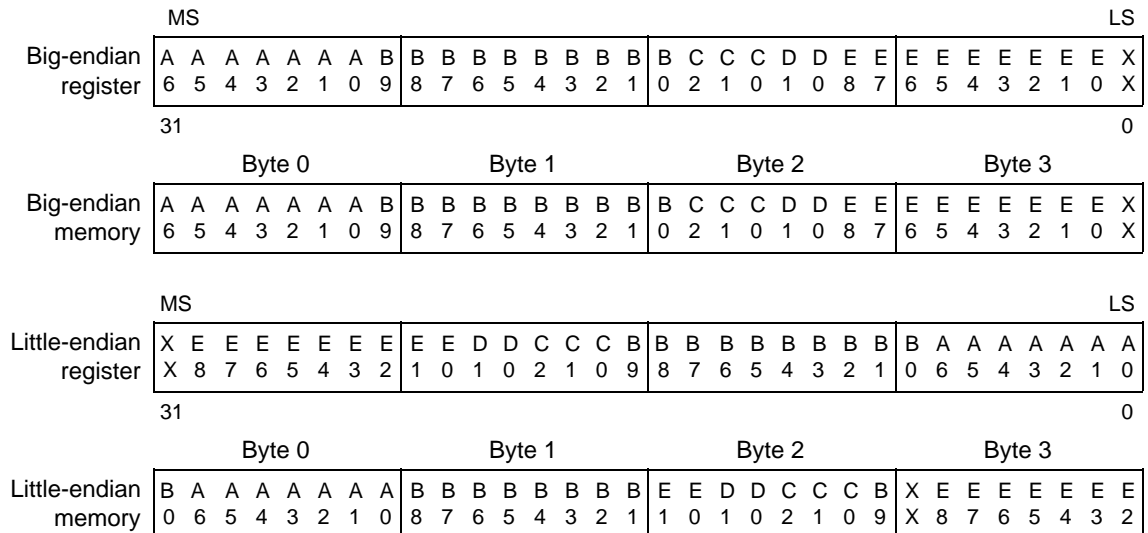
In big-endian mode, bit fields are packed into registers from most significant bit (MSB) to least significant bit (LSB) in the order in which they are defined and packed in memory from most significant byte (MSbyte) to least significant byte (LSbyte). For little-endian mode, bit fields are packed in registers from the LSB to the MSB in the order in which they are defined and packed in memory from LSbyte to MSbyte (see Figure 6–4).

In the following example of bit-field packing, assume these bit field definitions:

```
struct{
    int A:7
    int B:10
    int C:3
    int D:2
    int E:9
}x;
```

A0 represents the LSB of the field A; A1 represents the next LSB, etc. Again, storage of bit fields in memory is done with a byte-by-byte, rather than bit-by-bit, transfer.

Figure 6–4. Bit-Field Packing in Big-Endian and Little-Endian Formats



**Legend:** X = not used  
 MS = most significant  
 LS = least significant

### 6.2.3 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 6-35.

- ❑ 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
Link register	Contains the return address of a function call
Program counter	Contains the current address of code being executed

Table 6–4. Register Usage and Preservation Conventions

Register	Alias	Usage	Preserved by Function <sup>†</sup>
R0	A1	Argument register, return register, expression register	Callee
R1	A2	Argument register, return register, expression register	Callee
R2	A3	Argument register, expression register	Callee
R3	A4	Argument register, expression register	Callee
R4	V1	Expression register	Caller
R5	V2	Expression register	Caller
R6	V3	Expression register	Caller
R7	V4, AP	Expression register, argument pointer	Caller
R8	V5	Expression register	Caller
R9	V6	Expression register	Caller
R10	V7	Expression register	Caller
R11	V8	Expression register	Caller
R12	V9, IP	Expression register, instruction pointer	Callee
R13	SP	Stack pointer	Caller <sup>‡</sup>
R14	LR	Link register, expression register	Callee
R15	PC	Program counter	N/A
CPSR		Current program status register	Callee
SPSR		Saved program status register	Callee

<sup>†</sup> The caller function refers to the function making the function call. The callee function refers to the function being called.

<sup>‡</sup> 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

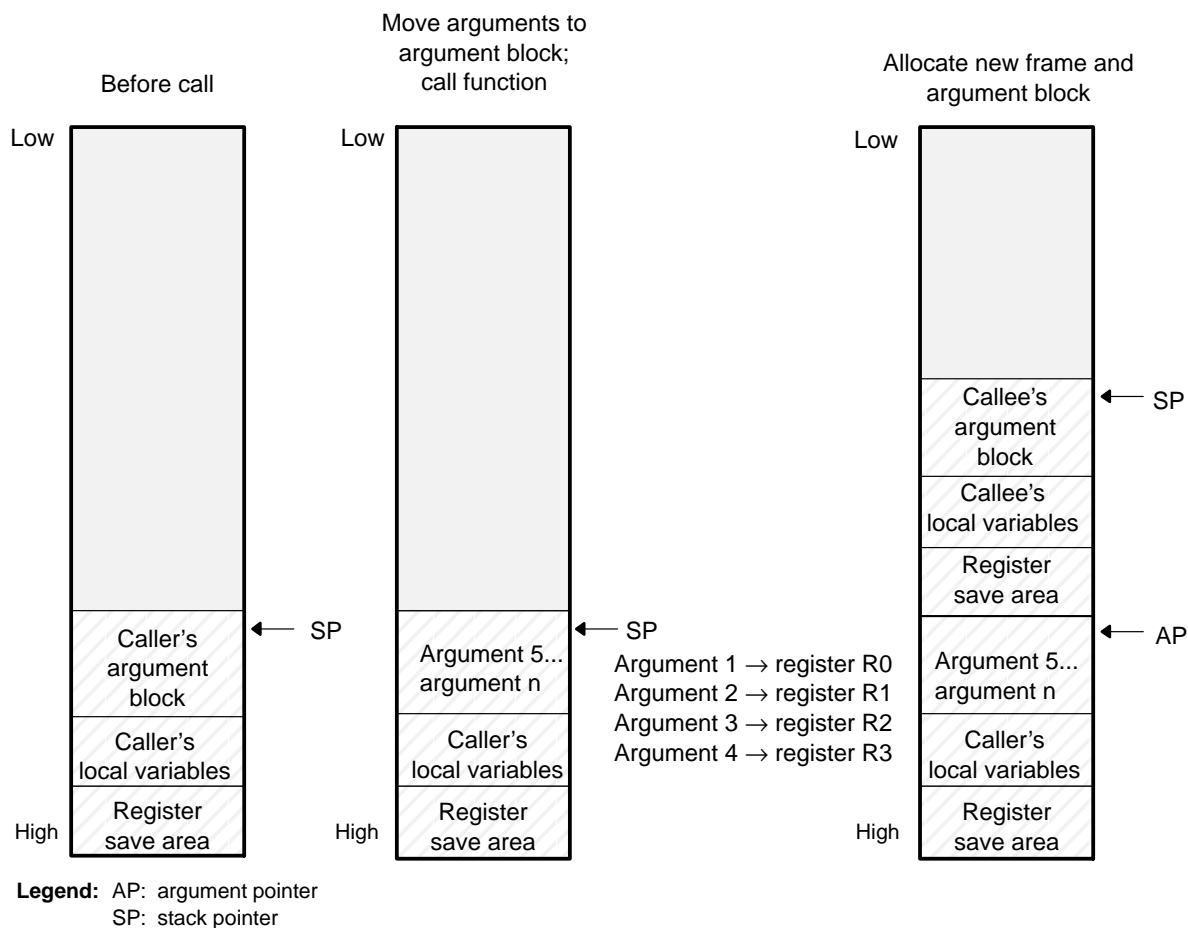
The C/C++ compiler imposes a strict set of rules on function calls. Except for special run-time-support functions, any function that calls or is called by a C/C++ function must follow these rules. Failure to adhere to these rules can disrupt the C/C++ environment and cause a program to fail.

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 R0–R3 and R12. 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–R11. 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 6–5 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 R0–R3. 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 6–5. Use of the Stack During a Function Call



### 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 R0–R3 and R12.)
- 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 R0–R3, 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) If arguments were stored onto the argument block in step 3, the caller reserves a word in the argument block for dual-state support. (See section 6.10, *Dual-State Interworking*, on page 6-41 for more information.)
- 5) 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–R11) and the link register (R14) 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 6-26.
- 3) If the function contains arguments on the stack that cannot be referenced by the stack pointer (SP or R13), the called function sets up a pointer to point to the first argument passed on the stack.

- 4) 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:

$$\text{size of all local variables} + \text{max} = \text{constant}$$

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

- 5) The called function executes the code for the function.
- 6) If the called function returns a value, it places the value in R0 (or R0 and R1 for floating-point double values).
- 7) If the called function returns a structure, it copies the structure to the memory block that the first argument, R0, points to. If the caller does not use the return value, R0 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).

- 8) The called function deallocates the frame and argument block by adding the constant computed in step 4.
- 9) The called function restores all registers saved in step 2.
- 10) The called function (*\_f*) loads the program counter (PC) with the return address.

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

```
_f:                                ; called function entry point
    STMFD  SP!, {V1, V2, V3, LR} ; save V1, V2, V3, and LR
    SUB    SP, SP, #16           ; allocate frame
    ...                               ; body of the function
    ADD    SP, SP, #16           ; deallocate frame
    LDMFD  SP!, {V1, V2, V3, PC} ; restore V1, V2, V3, and
                                ; store LR in the PC, causing
                                ; a return
```

### 6.4.3 Accessing Arguments and Local Variables

A function accesses its local nonregister variables indirectly through the stack pointer (SP or R13) and its stack arguments indirectly through the argument pointer (AP). If all stack arguments can be referenced with the SP, they are, and the AP is not reserved. The SP always points to the top of the stack (points to the most recently pushed value) and the AP points to the leftmost stack argument (the one closest to the top of the stack). For example:

```
LDR  A2 [SP, #4]    ; load local var from stack
LDR  A1 [AP, #0]    ; load argument from stack
```

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

### 6.4.4 Generating Long Calls (`-ml` Option) in 16-bit Mode

You can use the `-ml` option when compiling C/C++ code to produce long calls in 16-bit mode. You must also use the `-mt` (16-bit mode) and `-md` (no dual state) options, or `-ml` (long call) will have no effect.

The TMS470 C/C++ compiler normally uses the BL assembly instruction when making procedure calls in 16-bit mode. In object code, this is translated into two 16-bit BL instructions, each of which has 11 bits for the offset. Adding the two together, plus the assumed zero bit, gives only 23 bits of offset for a call. This is not enough when applications have widely dispersed areas of memory in which instructions reside.

The `-ml` option uses the BX instruction instead of the BL instruction. The BX instruction is less efficient. It requires a load instruction which uses a word of memory to load the destination address into a register, a move instruction to save the return address, and a BX instruction to make procedure calls.

## 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 6-23).
- ☐ Use inline assembly language embedded directly in the C/C++ source (see section 6.5.3 on page 6-25).
- ☐ Modify the assembly language code that the compiler produces (see section 6.5.4 on page 6-25).

### 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 6-13 and the calling conventions defined in section 6.4, *Function Structure and Calling Conventions*, on page 6-15. 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–R11)
- Stack pointer (SP or R13)

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 6-26.
- ☐ 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 6-17.

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. Double values are returned in R0 and R1, and structures are returned as described in step 2 of section 6.4.1, *How a Function Makes a Call*, on page 6-17. Any other values are returned in R0.
- ❑ 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 5.8, *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 or modules, any name that does not begin with an underscore or a dollar sign may be used safely without conflicting with a C/C++ identifier.

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

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

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

### Example 6–1. 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:
        LDR     r1, gvar_a
        LDR     r2, [r1, #0]
        ADD     r0, r0, r2
        STR     r0, [r1, #0]
        MOV     pc, lr
gvar_a   .field  _gvar, 32
```

In the C++ program in Example 6–1, the `extern "C"` declaration tells the compiler to use C naming conventions (i.e., no name mangling). When the linker resolves the `.global _asmfunc` reference, the corresponding definition in the assembly file will match.

The parameter `i` is passed in `R0`, and the result is returned in `R0`. `R1` holds the address of the global `gvar`. `R2` holds the value of `gvar` before adding the value of `i` to it.

## 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 6–2 shows how you can access a variable defined in .bss from C.

#### Example 6–2. 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 linker command file using a linker assignment statement. These constants are accessible from C/C++ only with the use of special operators.

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

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

You can use casts and `#defines` to ease the use of these symbols in your program, as in Example 6–3.

#### Example 6–3. 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 6–3, `int` is used. You can reference linker-defined symbols in a similar manner.

### 6.5.3 Using Inline Assembly Language

Within a C/C++ program, you can use the `asm` statement to insert a single line of assembly language into the assembly language file created by the compiler. A series of `asm` statements places sequential lines of assembly language into the compiler output with no intervening code. For more information, see section 5.6, *The asm Statement*, on page 5-13.

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

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

---

**Note: Using the asm Statement**

Keep the following in mind when using the `asm` statement:

- ☐ Be extremely careful not to disrupt the C/C++ environment. The compiler does not check or analyze the inserted instructions.
  - ☐ Inserting jumps or labels into C/C++ code can produce unpredictable results by confusing the register-tracking algorithms that the code generator uses.
  - ☐ Do not change the value of a C/C++ variable when using an `asm` statement.
  - ☐ Do not use the `asm` statement to insert assembler directives that change the assembly environment.
- 

### 6.5.4 Modifying Compiler Output

You can inspect and change the compiler's assembly language output by compiling the source and then editing the assembly output file before assembling it. The C/C++ `interlist` utility can help you inspect compiler output. (For information on the `interlist` utility, see section 2.10, *Using Interlist*, on page 2-40.)

## 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. With the exception of banked registers, register preservation must be explicitly handled by the interrupt routine.

All banked registers are automatically preserved by the hardware (except for interrupts that are reentrant. If you write interrupt routines that are reentrant, you must add code that preserves the interrupt's banked registers.) Each interrupt type has a set of banked registers. For information about the interrupt types, see section 5.7.4, *The INTERRUPT Pragma*, on page 5-17. For information about banked registers, see the *TMS470R1x User's Guide*.

### 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. Except for software interrupt routines, an interrupt routine must be declared with no arguments and must return void (see section 6.6.5, *Using Software Interrupts*). 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 5.7.4, *The INTERRUPT Pragma*, on page 5-17, 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 How to Map Interrupt Routines to Interrupt Vectors

The compiler tools include a file called `intvecs.asm`. This file contains assembly language directives that can be used to set up the TMS470's interrupt vectors with branches to your interrupt routines. Follow these steps to use this file:

- 1) Update `intvecs.asm` to include your interrupt routines. For each routine:
  - a) At the beginning of the file, add a `.global` directive that names the routine.
  - b) Modify the appropriate `.word` directive by replacing it with a branch to the name of your routine. The comments in the `intvecs.asm` file indicate which `.word` directive is associated with which interrupt vector.

---

**Note: Prefixing C/C++ Interrupt Routines**

Remember, if you are using C/C++ interrupt routines, you must use the proper linkname. So use `_c_intlIRQ` instead of `c_intlIRQ`, for example, for a function defined in C scope.

---

- 2) Assemble and link `intvecs.asm` with your applications code and with the compiler's linker control file (`lnk16.cmd` or `lnk32.cmd`). The control file contains a `SECTIONS` directive that maps the `.intvecs` section into the memory locations `0x00–0x1F`.

For example, if you have written a C interrupt routine for the IRQ interrupt called `c_intIRQ` and an assembly language routine for the FIQ interrupt called `tim1_int`, you should modify `intvecs.asm` as follows:

```
.global    _c_int00
.global    _c_intIRQ
.global    tim1_int

.sect     ".intvecs"
B         _c_int00      ; reset interrupt
.word     0             ; undefined instruction interrupt
.word     0             ; software interrupt
.word     0             ; abort (prefetch) interrupt
.word     0             ; abort (data) interrupt
.word     0             ; reserved
B         _c_intIRQ     ; IRQ interrupt
B         tim1_int      ; FIQ interrupt
```

### 6.6.5 Using Software Interrupts

A software interrupt is a synchronous exception generated by the execution of a particular instruction. Applications use software interrupts to request services from a protected system, such as an operating system, which can perform the services only while in a supervisor mode.

A C/C++ application can invoke a software interrupt by associating a software interrupt number with a function name through use of the `SWI_ALIAS` pragma and then calling the software interrupt as if it were a function. For information, see section 5.7.7, *The SWI\_ALIAS Pragma*, on page 5-20.

Since a call to the software interrupt function represents an invocation of the software interrupt, passing and returning data to and from a software interrupt is specified as normal function parameter passing with the following restriction:

All arguments passed to a software interrupt must reside in the four argument registers (R0–R3). No arguments can be passed by way of a software stack. Thus, only four arguments can be passed unless:

- ☐ Floating-point doubles are passed, in which case each double occupies two registers.
- ☐ Structures are returned, in which case the address of the returned structure occupies the first argument register.

The C/C++ compiler also treats the register usage of a called software interrupt the same as a called function. It assumes that all save-on-entry registers (R4–R11 and R13) are preserved by the software interrupt and that save-on-call registers (the remainder of the registers) can be altered by the software interrupt.



## 6.6.6 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. For example, refer to `c_int00` as `_c_int00`.
- ☐ When an interrupt occurs, the state of the processor changes to 32-bit. Therefore, the interrupt vectors must always contain branches to 32-bit state code. If the interrupt handler is a 16-bit state C/C++ function, the entry for the interrupt vector must be a branch to the interrupt routine's dual-state veneer (see section 6.10.1, *Level of Dual-State Support*).
- ☐ The FIQ, supervisor, abort, IRQ, and undefined modes have separate stacks that are not automatically set up by the C/C++ run-time environment. If you have interrupt routines in one of these modes, you must set up the software stack for that mode.
- ☐ Interrupt routines are not reentrant. If an interrupt routine enables interrupts of its type, it must save a copy of the return address and SPSR (the saved program status register) *before* doing so.
- ☐ Because a software interrupt is synchronous, the register saving conventions discussed in section 6.6.1, *Saving Registers During Interrupts*, on page 6-26 can be less restrictive as long as the system is designed for this. A software interrupt routine generated by the compiler, however, follows the conventions in section 6.6.1.

## 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 modulus, and floating-point operations.

There are two versions of each of the routines:

- ☐ A 16-bit version to be called only from the 16-BIS (bit instruction set) state
- ☐ A 32-bit version only to be called from the 32-BIS state

These routines do not follow the standard C/C++ calling conventions in that the naming and register conventions are not upheld. The compiler uses a preset naming convention for each of these routines, as described in section 6.7.1.

### 6.7.1 Naming Conventions

The names of the intrinsic run-time-support arithmetic and conversion routines are made up of two parts:

- ☐ The *prefix* indicates the type of the routine and the state from which it can be called. Table 6–5 lists the possible prefixes.
- ☐ The *root* indicates the operation performed by the routine. Table 6–6 lists the possible roots.

The prefixes from Table 6–5 are combined with the roots from Table 6–6 to form the function names listed in Table 6–7.

Table 6–5. Naming Convention Prefixes

Prefix	Type of Operation	State Called From
FD\$	Double-precision floating point	16-BIS
FD_	Double-precision floating point	32-BIS
FS\$	Single-precision floating point	16-BIS
FS_	Single-precision floating point	32-BIS
I\$	Signed integer	16-BIS
I_	Signed integer	32-BIS
U\$	Unsigned integer	16-BIS
U_	Unsigned integer	32-BIS

*Table 6–6. Summary of Run-Time-Support Arithmetic and Conversion Functions*

Root	Operation Performed	Type Usage			
		uint	int	single	double
ADD	Addition			√	√
CMP	Comparison			√	√
DIV	Division	√	√	√	√
MOD	Modulus	√	√		
MUL	Multiplication			√	√
SUB	Subtraction			√	√
TOI	Conversion to signed integer			√	√
TOU	Conversion to unsigned integer			√	√
TOFS	Conversion to single-precision floating point	√	√		√
TOFD	Conversion to double-precision floating point	√	√	√	

**Legend:** √ indicates that the combination is supported  
 uint unsigned integer  
 int signed integer  
 single single-precision floating point  
 double double-precision floating point

The source files for these functions are in the `rtsc.src` and `rtscpp.src` libraries. The source code has comments that describe the operation of the functions. You can extract, inspect, and modify any of these functions; be sure you follow the register usage conventions summarized in Table 6–7.

**Table 6–7. Run-Time-Support Function Register Usage Conventions**

Function	Inputs		Output	Source Filename
	Operand1	Operand2		
FS\$TOI	R0		R0	fs_toi16.asm
FS_TOI	R0		R0	fs_toi32.asm
FD\$TOI	R0:R1		R0	fd_toi16.asm
FD_TOI	R0:R1		R0	fd_toi32.asm
FS\$TOU	R0		R0	fs_tou16.asm
FS_TOU	R0		R0	fs_tou32.asm
FD\$TOU	R0:R1		R0	fd_tou16.asm
FD_TOU	R0:R1		R0	fd_tou32.asm
I\$TOFS	R0		R0	i_tofs16.asm
I_TOFS	R0		R0	i_tofs32.asm
U\$TOFS	R0		R0	u_tofs16.asm
U_TOFS	R0		R0	u_tofs32.asm
FD\$TOFS	R0:R1		R0	fd_tos16.asm
FD_TOFS	R0:R1		R0	fd_tos32.asm
I\$TOFD	R0		R0:R1	i_tofd16.asm
I_TOFD	R0		R0:R1	i_tofd32.asm
U\$TOFD	R0		R0:R1	u_tofd16.asm
U_TOFD	R0		R0:R1	u_tofd32.asm
FS\$TOFD	R0		R0:R1	fs_tod16.asm
FS_TOFD	R0		R0:R1	fs_tod32.asm
FS\$ADD	R0	R1†	R0	fs_add16.asm
FS_ADD	R0	R1†	R0	fs_add32.asm
FD\$ADD	R0:R1	R2:R3†	R0:R1	fd_add16.asm
FD_ADD	R0:R1	R2:R3†	R0:R1	fd_add32.asm

† Value in the register(s) is preserved.

Table 6–7. Run-Time-Support Function Usage Conventions (Continued)

Function	Inputs		Output	Source Filename
	Operand1	Operand2		
FS\$SUB	R0	R1	R0	fs_add16.asm
FS_SUB	R0	R1	R0	fs_add32.asm
FD\$SUB	R0:R1	R2:R3 <sup>†</sup>	R0:R1	fd_add16.asm
FD_SUB	R0:R1	R2:R3 <sup>†</sup>	R0:R1	fd_add32.asm
FS\$MUL	R0	R1 <sup>†</sup>	R0	fs_mul16.asm
FS_MUL	R0	R1 <sup>†</sup>	R0	fs_mul32.asm
FD\$MUL	R0:R1	R2:R3 <sup>†</sup>	R0:R1	fd_mul16.asm
FD_MUL	R0:R1	R2:R3 <sup>†</sup>	R0:R1	fd_mul32.asm
FS\$DIV	R0	R1	R0	fs_div16.asm
FS_DIV	R0	R1	R0	fs_div32.asm
FD\$DIV	R0:R1	R2:R3 <sup>†</sup>	R0:R1	fd_div16.asm
FD_DIV	R0:R1	R2:R3 <sup>†</sup>	R0:R1	fd_div32.asm
I\$DIV	R0	R1	R1	i_div16.asm
I_DIV	R0	R1	R1	i_div32.asm
U\$DIV	R0	R1	R1	u_div16.asm
U_DIV	R0	R1	R1	u_div32.asm
I\$MOD	R0	R1	R0	i_mod16.asm
I_MOD	R0	R1	R0	i_mod32.asm
U\$MOD	R0	R1	R0	u_mod16.asm
U_MOD	R0	R1	R0	u_mod32.asm
FS\$CMP	R0 <sup>†</sup>	R1 <sup>†</sup>	CPSR	fs_cmp16.asm
FS_CMP	R0 <sup>†</sup>	R1 <sup>†</sup>	CPSR	fs_cmp32.asm
FD\$CMP	R0:R1 <sup>†</sup>	R2:R3 <sup>†</sup>	CPSR	fd_cmp16.asm
FD_CMP	R0:R1 <sup>†</sup>	R2:R3 <sup>†</sup>	CPSR	fd_cmp32.asm

<sup>†</sup> Value in the register(s) is preserved.

## 6.8 Built-In Functions

Built-in functions are predefined by the compiler. They can be called like a regular function, but they do not require a prototype or definition. The compiler supplies the proper prototype and definition.

The TMS470 compiler supports the following built-in functions:

- ❑ The `__curpc` function, which returns the value of the program counter where it is called. The syntax of the function is:

```
void *__curpc(void);
```

- ❑ The `__run_address_check` function, which returns TRUE if the code performing the call is located at its run-time address, as assigned by the linker. Otherwise, FALSE is returned. The syntax of the function is:

```
int __run_address_check(void);
```

## 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 `-c` or `-cr` linker function option and include the run-time library as one of the linker input files.

When C/C++ programs are linked, the linker sets the entry point value in the executable output module to the symbol `_c_int00`. The `_c_int00` function performs the following tasks to initialize the C/C++ environment:

- 1) Switches from system mode to user mode, 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 (`-cr` option), a loader performs this step before the program runs (it is not performed by the boot routine). For more information, see section 6.9.2, *Automatic Initialization of Variables*.
- 3) Executes the global constructors found in the `.pinit` section. For more information, see section 6.9.3, *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 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` linker option on the linker 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.

### 6.9.2 Automatic Initialization of Variables

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

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

---

**Note: Initializing Variables**

In ANSI C, global and static variables that are not explicitly initialized must be set to 0 before program execution. The C/C++ compiler does not perform any preinitialization of uninitialized variables. You must explicitly initialize any variable that must have an initial value of 0.

---

### 6.9.3 Global Constructors

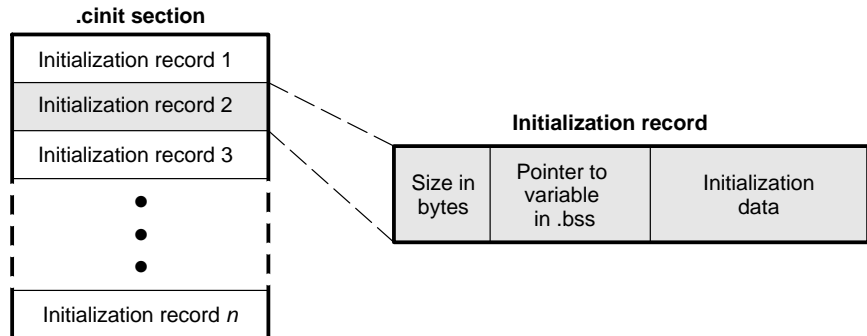
All global C++ variables that have constructors must have their constructor called before `main()`. The compiler builds a table of global constructor addresses that must be called, in order, before `main()` in a section called `.pinit`. The linker combines the `.pinit` section from each input file to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



### 6.9.4 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 6–6 shows the format of the `.cinit` section and the initialization records.

Figure 6–6. *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 (32 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 6–4 shows two initialized variables defined in C. Example 6–4 (b) shows the corresponding initialization table.

Example 6–4. 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)

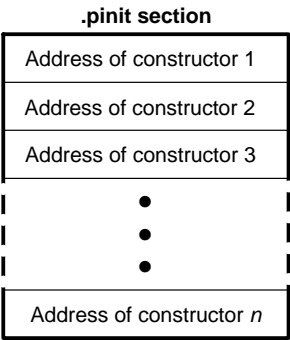
```
.sect      ".cinit"      ; Initialization section
* Initialization record for variable i
  .align    4             ; align on word boundary
  .field    4,32          ; length of data (1 word)
  .field    _i+0,32       ; address of i
  .field    23,32         ; _i @ 0

* Initialization record for variable a
  .sect      ".cinit"
  .align    4             ; align on word boundary
  .field    IR1,32        ; Length of data (5 words)
  .field    _a+0,32       ; Address of a[]
  .field    1,32          ; _a[0] @ 0
  .field    2,32          ; _a[1] @ 32
  .field    3,32          ; _a[2] @ 64
  .field    4,32          ; _a[3] @ 96
  .field    5,32          ; _a[4] @ 128
IR1:  .set    20           ; set length symbol
```

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 6–7). The constructors appear in the table in the order they must be executed.

Figure 6–7. Format of the .pinit Section



When you use the `-c` or `-cr` option, the linker 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 `-c` or `-cr` linker option causes the linker 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.

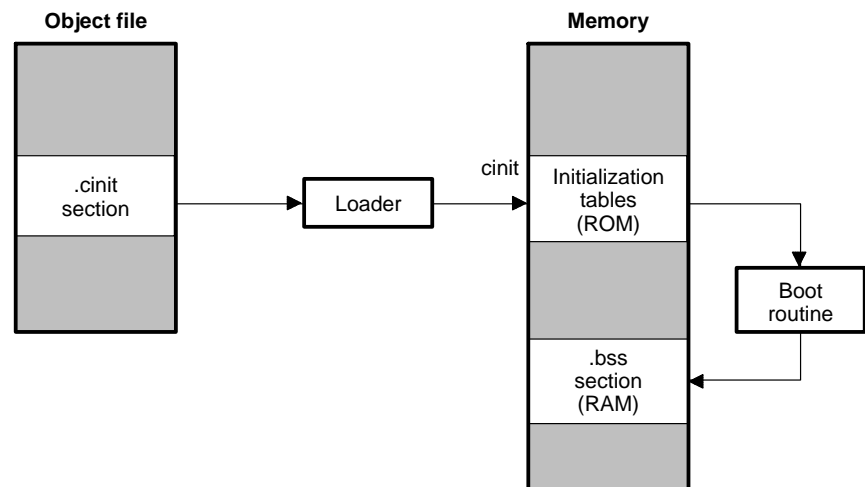
### 6.9.5 Autoinitialization of Variables at Run Time

Autoinitializing variables at run time is the default model for autoinitialization. To use this method, invoke the linker with the `-c` option.

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

Figure 6–8 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into ROM.

Figure 6–8. Autoinitialization at Run Time



### 6.9.6 Autoinitialization of Variables at Load Time

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

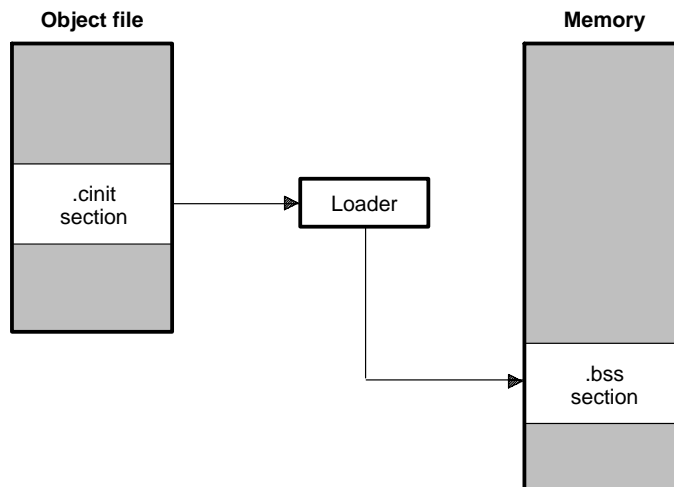
When you use the `-cr` linker option, the linker sets the `STYP_COPY` bit in the `.cinit` section's header. This tells the loader not to load the `.cinit` section into memory. (The `.cinit` section occupies no space in the memory map.) The linker also sets the `cinit` symbol to `-1` (normally, `cinit`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

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

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

Figure 6–9 illustrates the RAM model of autoinitialization.

*Figure 6–9. Autoinitialization at Load Time*



Whether or not you use the `-c` or `-cr` linker option, the `.pinit` section is always loaded and processed at run time.

## 6.10 Dual-State Interworking

The TMS470 is a unique processor in that it gives you the performance of a 32-bit architecture with the code density of a 16-bit architecture. It does this by supporting a 16-bit instruction set and a 32-bit instruction set and allowing switching dynamically between the two sets.

The instruction set that the TMS470 processor uses is determined by the state of the processor. The processor can be in 32-BIS (bit instruction set) state or 16-BIS state at any given time. The compiler allows you to specify whether a module should be compiled in 32- or 16-BIS state and allows functions compiled in one state to call functions compiled in the other state.

### 6.10.1 Level of Dual-State Support

By default, the compiler allows dual-state interworking between functions. However, the compiler allows you to alter the level of support to meet your particular needs.

In dual-state interworking, it is the called function's responsibility to handle the proper state changes required by the calling function. It is the calling function's responsibility to handle the proper state changes required to *indirectly* call a function (call it by address). Therefore, a function supports dual-state interworking if it provides the capability for functions requiring a state change to *directly* call the function (call it by name) and provides the mechanism to indirectly call functions involving state changes.

If a function does not support dual-state interworking, it cannot be called by functions requiring a state change and cannot indirectly call functions that support dual-state interworking. Regardless of whether a function supports dual-state interworking or not, it can directly or indirectly call certain functions:

- ☐ Directly call a function in the same state
- ☐ Directly call a function in a different state if that function supports dual-state interworking
- ☐ Indirectly call a function in the same state if that function does not support dual-state interworking

Given this definition of dual-state support, the TMS470 C/C++ compiler offers three levels of support. Use Table 6–8 to determine the best level of support to use for your code.

*Table 6–8. Selecting a Level of Dual-State Support*

If your code...	Use this level of support...
Requires few state changes	Default
Requires many state changes	Optimized
Requires no state changes and has frequent indirect calls	None

Here is detailed information about each level of support:

- ☐ **Default.** Full dual-state interworking is supported. For each function that supports full dual-state interworking, the compiler generates code that allows functions requiring a state change to call the function, whether it is ever used or not. This code is placed in a different section from the section the actual function is in. If the linker determines that this code is never referenced, it does not link it into the final executable image. However, the mechanism used with indirect calls to support dual-state interworking is integrated into the function and cannot be removed by the linker, even if the linker determines that the mechanism is not needed.
- ☐ **Optimized.** Optimized dual-state interworking provides no additional functionality over the default level but optimizes the dual-state support code (in terms of code size and execution speed) for the case where a state change is required. It does this optimization by integrating the support into the function. Use the optimized level of support only when you know that a majority of the calls to this function require a state change. Even if the dual-state support code is never used, the linker cannot remove the code because it is integrated into the function. To specify this level of support, use the `DUAL_STATE` pragma. See section 5.7.2, *The DUAL\_STATE Pragma*, on page 5-15 for more information.

❑ **None.** Dual-state interworking is disabled. This level is invoked with the `-md shell` option. Functions with this support can *directly* call the following functions:

- Functions compiled in the same state
- Functions in a different state that support dual-state interworking

Functions with this support level can *indirectly* call only functions that do not require a state change and do not support dual-state interworking. Because functions with this support level do not provide dual-state interworking, they cannot be called by a function requiring a state change.

Use this support level if you do not require dual-state interworking, have frequent indirect calls, and cannot tolerate the additional code size or speed incurred by the indirect calls supporting dual-state interworking.

When a program does not require any state changes, the only difference between specifying no support and default support is that indirect calls are more complex in the default support level.

## 6.10.2 Implementation

Dual-state support is implemented by providing an alternate entry point for a function. This alternate entry point is used by functions requiring a state change. Dual-state support handles the change to the correct state and, if needed, changes the function back to the state of the caller when it returns. Also, indirect calls set up the return address so that once the called function returns, the state can be reliably changed back to that of the caller.

### 6.10.2.1 Naming Conventions for Entry Points

The TMS470 compiler reserves the name space of all identifiers beginning with an underscore (`_`) or a dollar sign (`$`). In this dual-state support scheme, all 32-BIS state entry points begin with an underscore, and all 16-BIS state entry points begin with a dollar sign. All other compiler-generated identifiers, which are independent of the state of the processor, begin with an underscore. By this convention, all direct calls within a 16-bit function refer to the entry point beginning with a dollar sign and all direct calls within a 32-bit function refer to the entry point beginning with an underscore.

### 6.10.2.2 Indirect Calls

Addresses of functions taken in 16-BIS state use the address of the 16-BIS state entry point to the function (with bit 0 of the address set). Likewise, addresses of functions taken in 32-BIS state use the address of the 32-BIS state entry point (with bit 0 of the address cleared). Then all indirect calls are performed by loading the address of the called function into a register and executing the branch and exchange (BX) instruction. This automatically changes the state and ensures that the code works correctly, regardless of what state the address was in when it was taken.

The return address must also be set up so that the state of the processor is consistent and known upon return. Bit 0 of the address is tested to determine if the BX instruction invokes a state change. If it does not invoke a state change, the return address is set up for the state of the function. If it does invoke a change, the return address is set up for the alternate state and code is executed to return to the function's state.

Because the entry point into a function depends upon the state of the function that takes the address, it is more efficient to take the address of a function when in the same state as that function. This ensures that the address of the actual function is used, not its alternate entry point. Because the indirect call can invoke a state change itself, entering a function through its alternate entry point, even if calling it from a different state, is unnecessary.

Example 6–5 shows `sum( )` calling `max( )` with code that is compiled for the 16-BIS state and supports dual-state interworking. The `sum( )` function is compiled with the `-mt` option, which creates 16-bit instructions. Example 6–6 shows the same function call with code that is compiled for the 32-BIS state and supports dual-state interworking. Function `max( )` is compiled without the `-mt` option, creating 32-bit instructions.



*Example 6–5. Code Compiled for 16-BIS State: sum( )**(a) C program*

```

int total = 0;

sum(int val1, int val2)
{
    int val = max(val1, val2);

    total += val;
}

```

*(b) 16-bit assembly program*

```

;*****
;* FUNCTION VENEER: _sum                                     *
;*****
_sum:
    .state32
    STMFD sp!, {lr}
    ADD    lr, pc, #1
    BX     lr
    .state16
    BL     $sum
    BX     pc
    NOP
    .state32
    LDMFD sp!, {pc}
    .state16
    .sect  ".text"
    .global $sum

;*****
;* FUNCTION DEF: $sum                                       *
;*****
$sum:
    PUSH    {LR}
    BL      $max
    LDR     A2, CON1 ; {_total+0}
    LDR     A3, [A2, #0]
    ADD     A1, A1, A3
    STR     A1, [A2, #0]
    POP     {PC}

;*****
;* CONSTANT TABLE                                         *
;*****
    .sect  ".text"
    .align 4
CON1: .field _total, 32

```

**Example 6–6. Code Compiled for 32-BIS State: max( )****(a) C program**

```
int max(int a, int b)
{
    return a < b ? b : a;
}
```

**(b) 32-bit assembly program**

```
;*****
;* FUNCTION VENEER: $max
;* *****
$max:
    .state16
    BX     pc
    NOP
    .state32
    B      _max
    .text

    .global _max

:*****
;* FUNCTION DEF: _max
;* *****
_max
    CMP    A1, A2
    MOVLE  A1, A2
    BX     LR
```

Since `sum( )` is a 16-bit function, its entry point is `$sum`. Because it was compiled for dual-state interworking, an alternate entry point, `_sum`, located in a different section is included. All calls to `sum( )` requiring a state change use the `_sum` entry point.

The call to `max( )` in `sum( )` references `$max`, because `sum( )` is a 16-bit function. If `max( )` were a 16-bit function, `sum( )` would call the actual entry point for `max( )`. However, since `max( )` is a 32-bit function, `$max` is the alternate entry point for `max( )` and handles the state change required by `sum( )`.

# 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 TMS470R1x 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 TMS470 run-time-support library includes routines that give you processor-specific commands and direct C language I/O requests.

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

Topic	Page
7.1 Libraries .....	7-2
7.2 The C I/O Functions .....	7-4
7.3 Header Files .....	7-16
7.4 Summary of Run-Time-Support Functions and Macros .....	7-29
7.5 Description of Run-Time-Support Functions and Macros .....	7-39

## 7.1 Libraries

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

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

When a library is linked, the linker includes only those library members required to resolve undefined references. For more information about linking, see the linker description chapter of the *TMS470R1x Assembly Language Tools User's Guide*.

### 7.1.1 Nonstandard Header Files in `rtsc.src` and `rtscpp.src`

The `rtsc.src` and `rtscpp.src` files 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 `470cio.h` file includes low-level, target-specific C I/O macro definitions. If necessary, you can customize `470cio.h`.
- ☐ 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 using the archiver to extract the appropriate source file or files from `rtsc.src` or `rtscpp.src`. For example, the following command extracts two source files:

```
ar470 -x rtsc.src atoi.c strcpy.c
```

To modify a function, extract the source as in the previous example. Make the required changes to the code, recompile, and reinstall the new object file(s) into the library. The following example shows how you can reinstall the updated code back into the library:

```
ar470 -r rtsc.src atoi.c strcpy.c
```

Then, use one of these methods to build a new object library based on the updated source files:

- ☐ Use the library-build utility to rebuild the entire library (see Chapter 8).
- ☐ Recompile the updated source files and reinstall the object files into the object library.

For more information about the archiver, see the archiver description in the *TMS470R1x Assembly Language Tools User's Guide*.

## 7.1.3 Building a Library With Different Options

You can create a new library from `rtsc.src` or `rtscpp.src` by using the library-build utility, `mk470`. For example, use this command to build an optimized run-time-support library:

```
mk470 --u -o2 -mf rtsc.src -l rtsf.lib
```

The `--u` option tells the `mk470` utility to use the header files in the current directory, rather than extracting them from the source archive. The new library is compatible with any code compiled for the TMS470. The use of the optimizer (`-o2`) and generate fast code (`-mf`) options does not affect compatibility with code compiled without these options. For more information about building libraries, see Chapter 8, *Library-Build Utility*.

## 7.2 The C I/O Functions

The C I/O functions make it possible to access the host's operating system to perform I/O (using the debugger). For example, `printf` statements executed in a 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, *Linker 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");
    fprintf(fid, "Hello, world\n");
    fclose(fid);

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

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

```
cl470 main.c -z -l rtsc_32.lib -o main.out
```

Executing `main.out` under the TMS470 debugger on a SPARC™ 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.

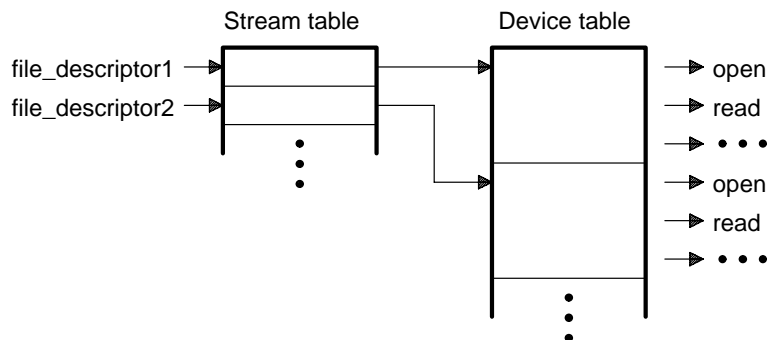
The high-level functions are the standard C library of stream I/O routines (`printf`, `scanf`, `fopen`, `getchar`, etc.). These routines map an I/O request to one or more of the I/O commands that are handled by the low-level shell.

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.

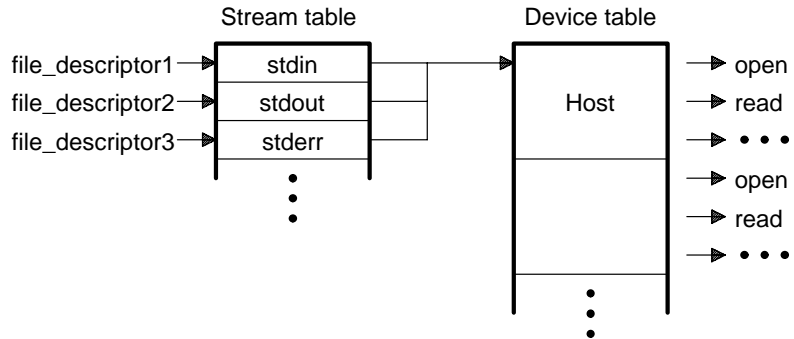
The data structures interact as shown in Figure 7–1.

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 7-14. 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 pre-defined 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 7-5. 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	<i>Close File or Device for I/O</i>				
Syntax for C	<pre>#include &lt;stdio.h&gt; #include &lt;file.h&gt;  int close(int file_descriptor);</pre>				
Syntax for C++	<pre>#include &lt;cstdio&gt; #include &lt;file.h&gt;  int std::close(int file_descriptor);</pre>				
Description	<p>The close function closes the device or file associated with <i>file_descriptor</i>.</p> <p>The <i>file_descriptor</i> is the stream number assigned by the low-level routines that is associated with the opened device or file.</p>				
Return Value	<p>The function returns one of the following values:</p> <table><tr><td>0</td><td>if successful</td></tr><tr><td>-1</td><td>if fails</td></tr></table>	0	if successful	-1	if fails
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:

<b>SEEK_SET</b>	(0x0000) Beginning of file
<b>SEEK_CUR</b>	(0x0001) Current value of the file position indicator
<b>SEEK_END</b>	(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:

- $\neq -1$  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 7-5.

---

**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:

<b>name</b>	String for device name
<b>flags</b>	Specifies whether the device supports multiple streams or not
<b>function pointers</b>	Pointers to the device-level functions: <ul style="list-style-type: none"><li><input type="checkbox"/> <code>CLOSE</code></li><li><input type="checkbox"/> <code>LSEEK</code></li><li><input type="checkbox"/> <code>OPEN</code></li><li><input type="checkbox"/> <code>READ</code></li><li><input type="checkbox"/> <code>RENAME</code></li><li><input type="checkbox"/> <code>WRITE</code></li><li><input type="checkbox"/> <code>UNLINK</code></li></ul>

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 7-7.



- 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	linkage.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 7.3.10 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 7-29, 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:

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

#elif 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 *isxxx* (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 *toxxx* (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 7-30 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 TMS470 is described in section 7.2, *The C/I/O Functions*, on page 7-4.

### 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 TMS470R1x'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	–32 768	Minimum value for a short int
SHRT_MAX	32 767	Maximum value for a short int
USHRT_MAX	65 535	Maximum value for an unsigned short int
INT_MIN	(–INT_MAX – 1)	Minimum value for an int
INT_MAX	2 147 483 647	Maximum value for an int
UINT_MAX	4 294 967 295	Maximum value for an unsigned int
LONG_MIN	(–LONG_MAX – 1)	Minimum value for a long int
LONG_MAX	2 147 483 647	Maximum value for a long int
ULONG_MAX	4 294 967 295	Maximum value for an unsigned long int
LLONG_MIN	(–LLONG_MAX – 1)	Minimum value for a long long int
LLONG_MAX	9 223 372 036 854 775 807	Maximum value for a long long int
ULLONG_MAX	18 446 744 073 709 551 615	Maximum value for an unsigned long 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	6	Number of decimal digits of precision for a float, double, or long double
DBL_DIG	15	
LDBL_DIG	15	
FLT_MANT_DIG	24	Number of base-FLT_RADIX digits in the mantissa of a float, double, or long double
DBL_MANT_DIG	53	
LDBL_MANT_DIG	53	
FLT_MIN_EXP	–125	Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized float, double, or long double
DBL_MIN_EXP	–1021	
LDBL_MIN_EXP	–1021	
FLT_MAX_EXP	128	Maximum negative integer such that FLT_RADIX raised to that power minus 1 is a representable finite float, double, or long double
DBL_MAX_EXP	1024	
LDBL_MAX_EXP	1024	
FLT_EPSILON	1.19209290e–07	Minimum positive float, double, or long double number $x$ such that $1.0 + x \neq 1.0$
DBL_EPSILON	2.22044605e–16	
LDBL_EPSILON	2.22044605e–16	
FLT_MIN	1.17549435e–38	Minimum positive float, double, or long double
DBL_MIN	2.22507386e–308	
LDBL_MIN	2.22507386e–308	
FLT_MAX	3.40282346e+38	Maximum float, double, or long double
DBL_MAX	1.79769313e+308	
LDBL_MAX	1.79769313e+308	
FLT_MIN_10_EXP	–37	Minimum negative integers such that 10 raised to that power is in the range of normalized floats, doubles, or long doubles
DBL_MIN_10_EXP	–307	
LDBL_MIN_10_EXP	–307	
FLT_MAX_10_EXP	38	Maximum positive integers such that 10 raised to that power is in the range of representable finite floats, doubles, or long doubles
DBL_MAX_10_EXP	308	
LDBL_MAX_10_EXP	308	

**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, a structure type of the type of the value returned by the `imaxdiv` function
- ❑ The `intmax_t` type, an integer type large enough to represent any value of any signed integer type
- ❑ The `uintmax_t` type, 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 `printf` and `fscanf` macros. For example, three `printf` 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 Language – 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 Function Calls as near or far (linkage.h)

The linkage.h header declares macros that determine how code and data in the run-time support library is accessed.. Depending on the value of the `_FAR_RTS` macro, the `_CODE_ACCESS` macro is set to force calls to run-time-support functions to be either user default, near or far. The `_FAR_RTS` macro is set according to the use of the `-mr` compiler option.

The `_DATA_ACCESS` macro is set to always be far. The `_IDECL` macro determines how inline functions are declared.

All header files that define functions or data declare `#include <linkage.h>`. Functions are modified with `_CODE_ACCESS`, for example:

```
extern _CODE_ACCESS void    exit(int _status);
```

Data is modified with `_DATA_ACCESS`, for example:

```
extern _DATA_ACCESS unsigned char _ctypes[];
```

### 7.3.9 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 7-31.



### 7.3.10 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 7-32 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.

### 7.3.11 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 7-32.

### 7.3.12 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.

### 7.3.13 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 *intN\_t* and of the unsigned form *uintN\_t*
- ☐ Integer types with at least certain specified widths of the signed form *int\_leastN\_t* and of the unsigned form *uint\_leastN\_t*
- ☐ Fastest integer types with at least certain specified widths of the signed form *int\_fastN\_t* and of the unsigned form *uint\_fastN\_t*
- ☐ Signed, *intptr\_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 *INTN\_C(value)* macro expands to a signed integer constant with the specified value and type *int\_leastN\_t*. The unsigned *UINTN\_C(value)* macro expands to an unsigned integer constant with the specified value and type *uint\_leastN\_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)*.

### 7.3.14 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 `stddef.h/cstddef`.
- ☐ The *fpos\_t* 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 `stddef.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 7-32.

### 7.3.15 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 linker with the `-heap` 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 7-35.

### 7.3.16 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 `strxxx` 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 `memxxx`.

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 7-36.

### 7.3.17 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 7-38.

You can 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.

### 7.3.18 Exception Handling (exception and stdexcept)

Exception handling is not supported. The `exception` and `stdexcept` include files, which are for C++ only, are empty.

### 7.3.19 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.

### 7.3.20 Run-Time Type Information (typeid)

The `typeid` 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 TMS470R1x 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 7-39. 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 <b>assert</b> (int expression);	Inserts diagnostic messages into programs	7-42

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

Function	Description	Page
int <b>isalnum</b> (int c);	Tests c to see if it is an alphanumeric ASCII character	7-62
int <b>isalpha</b> (int c);	Tests c to see if it is an alphabetic ASCII character	7-62
int <b>isascii</b> (int c);	Tests c to see if it is an ASCII character	7-62
int <b>iscntrl</b> (int c);	Tests c to see if it is a control character	7-62
int <b>isdigit</b> (int c);	Tests c to see if it is a numeric character	7-62
int <b>isgraph</b> (int c);	Tests c to see if it is any printing character except a space	7-62
int <b>islower</b> (int c);	Tests c to see if it is a lowercase alphabetic ASCII character	7-62
int <b>isprint</b> (int c);	Tests c to see if it is a printable ASCII character (including spaces)	7-62
int <b>ispunct</b> (int c);	Tests c to see if it is an ASCII punctuation character	7-62
int <b>isspace</b> (int c);	Tests c to see if it is an ASCII spacebar, tab (horizontal or vertical), carriage return, formfeed, or newline character	7-62
int <b>isupper</b> (int c);	Tests c to see if it is an uppercase alphabetic ASCII character	7-62
int <b>isxdigit</b> (int c);	Tests c to see if it is a hexadecimal digit	7-62
int <b>toascii</b> (int c);	Masks c into a legal ASCII value	7-96
int <b>tolower</b> (int c);	Converts c to lowercase if it is uppercase	7-96
int <b>toupper</b> (int c);	Converts c to uppercase if it is lowercase	7-96

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



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

Function	Description	Page
double <b>acos</b> (double x);	Returns the arc cosine of x	7-40
double <b>asin</b> (double x);	Returns the arc sine of x	7-41
double <b>atan</b> (double x);	Returns the arc tangent of x	7-43
double <b>atan2</b> (double y, double x);	Returns the inverse tangent of y/x	7-43
double <b>ceil</b> (double x);	Returns the smallest integer greater than or equal to x; expands to inline if <code>-x</code> option is used	7-46
double <b>cos</b> (double x);	Returns the cosine of x	7-48
double <b>cosh</b> (double x);	Returns the hyperbolic cosine of x	7-48
double <b>exp</b> (double x);	Returns the exponential function of x; expands inline unless <code>-x0</code> option is used	7-51
double <b>fabs</b> (double x);	Returns the absolute value of x	7-52
double <b>floor</b> (double x);	Returns the largest integer less than or equal to x; expands inline if <code>-x</code> option is used	7-54
double <b>fmod</b> (double x, double y);	Returns the floating-point remainder of x/y; expands inline if <code>-x</code> option is used	7-55
double <b>frexp</b> (double value, int *exp);	Breaks value into a normalized fraction and an integer power of 2	7-58
double <b>ldexp</b> (double x, int exp);	Multiplies x by an integer power of 2	7-63
double <b>log</b> (double x);	Returns the natural logarithm of x	7-64
double <b>log10</b> (double x);	Returns the base-10 (common) logarithm of x	7-65
double <b>modf</b> (double value, double *iptr);	Breaks value into a signed integer and a signed fraction	7-71
double <b>pow</b> (double x, double y);	Returns x raised to the power y	7-72
double <b>sin</b> (double x);	Returns the sine of x	7-79
double <b>sinh</b> (double x);	Returns the hyperbolic sine of x	7-79
double <b>sqrt</b> (double x);	Returns the nonnegative square root of x	7-80
double <b>tan</b> (double x);	Returns the tangent of x	7-94
double <b>tanh</b> (double x);	Returns the hyperbolic tangent of x	7-94

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

Macro or Function	Description	Page
int <b>setjmp</b> (jmp_buf env);	Saves calling environment for later use by longjmp function; expands inline if -x option is used	7-77
void <b>longjmp</b> (jmp_buf env, int _val);	Uses jmp_buf argument to restore a previously saved program environment	7-77

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

Macro	Description	Page
type <b>va_arg</b> (_ap, type);	Accesses the next argument of type <i>type</i> in a variable-argument list	7-97
void <b>va_end</b> (_ap);	Resets the calling mechanism after using va_arg	7-97
void <b>va_start</b> (_ap, parmN);	Initializes ap to point to the first operand in the variable-argument list	7-97

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

Function	Description	Page
void <b>clearerr</b> (FILE *_fp);	Clears the EOF and error indicators for the stream that _fp points to	7-47
int <b>fclose</b> (FILE *_fp);	Flushes the stream that _fp points to and closes the file associated with that stream	7-52
int <b>feof</b> (FILE *_fp);	Tests the EOF indicator for the stream that _fp points to	7-52
int <b>ferror</b> (FILE *_fp);	Tests the error indicator for the stream that _fp points to	7-53
int <b>fflush</b> (register FILE *_fp);	Flushes the I/O buffer for the stream that _fp points to	7-53
int <b>fgetc</b> (register FILE *_fp);	Reads the next character in the stream that _fp points to	7-53
int <b>fgetpos</b> (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	7-53
char <b>*fgets</b> (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	7-54
FILE <b>*fopen</b> (const char *_fname, const char *_mode);	Opens the file that _fname points to; _mode points to a string describing how to open the file	7-55
int <b>fprintf</b> (FILE *_fp, const char *_format, ...);	Writes to the stream that _fp points to	7-55

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

Function	Description	Page
int <b>fputc</b> (int _c, register FILE *_fp);	Writes a single character, _c, to the stream that _fp points to	7-56
int <b>fputs</b> (const char *_ptr, register FILE *_fp);	Writes the string pointed to by _ptr to the stream pointed to by _fp	7-56
size_t <b>fread</b> (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	7-56
FILE <b>*freopen</b> (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	7-57
int <b>fscanf</b> (FILE *_fp, const char *_fmt, ...);	Reads formatted input from the stream that _fp points to	7-58
int <b>fseek</b> (register FILE *_fp, long _offset, int _ptrname);	Sets the file position indicator for the stream that _fp points to	7-58
int <b>fsetpos</b> (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.	7-59
long <b>ftell</b> (FILE *_fp);	Obtains the current value of the file position indicator for the stream that _fp points to	7-59
size_t <b>fwrite</b> (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	7-59
int <b>getc</b> (FILE *_p);	Reads the next character in the stream that _fp points to	7-60
int <b>getchar</b> (void);	A macro that calls fgetc() and supplies stdin as the argument	7-60
char <b>*gets</b> (char *_ptr);	Performs the same function as fgets() using stdin as the input stream	7-61
void <b> perror</b> (const char *_s);	Maps the error number in _s to a string and prints the error message	7-71
int <b>printf</b> (const char *_format, ...);	Performs the same function as fprintf() but uses stdout as its output stream	7-72
int <b>putc</b> (int _x, FILE *_fp);	A macro that performs like fputc()	7-72
int <b>putchar</b> (int _x);	A macro that calls fputc() and uses stdout as the output stream	7-73
int <b>puts</b> (const char *_ptr);	Writes the string pointed to by _ptr to stdout	7-73

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

Function	Description	Page
int <b>remove</b> (const char *_file);	Causes the file with the name pointed to by _file to be no longer available by that name	7-75
int <b>rename</b> (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	7-76
void <b>rewind</b> (register FILE *_fp);	Sets the file position indicator for the stream pointed to by _fp to the beginning of the file	7-76
int <b>scanf</b> (const char *_fmt, ...);	Performs the same function as fscanf() but reads input from stdin	7-76
void <b>setbuf</b> (register FILE *_fp, char *_buf);	Returns no value. setbuf() is a restricted version of setvbuf() and defines and associates a buffer with a stream	7-77
int <b>setvbuf</b> (register FILE *_fp, register char *_buf, register int _type, register size_t _size);	Defines and associates a buffer with a stream	7-78
int <b>sprintf</b> (char *_string, const char *_format, ...);	Performs the same function as fprintf() but writes to the array that _string points to	7-80
int <b>sscanf</b> (const char *_str, const char *_fmt, ...);	Performs the same function as fscanf() but reads from the string that _str points to	7-81
FILE * <b>tmpfile</b> (void);	Creates a temporary file	7-95
char * <b>tmpnam</b> (char *_s);	Generates a string that is a valid filename (that is, the filename is not already being used)	7-96
int <b>ungetc</b> (int _c, register FILE *_fp);	Pushes the character specified by _c back into the input stream pointed to by _fp	7-97
int <b>vfprintf</b> (FILE *_fp, const char *_format, va_list _ap);	Performs the same function as fprintf() but replaces the argument list with _ap	7-98
int <b>vprintf</b> const char *_format, va_list _ap);	Performs the same function as printf() but replaces the argument list with _ap	7-98
int <b>vsprintf</b> (char *_string, const char *_format, va_list _ap);	Performs the same function as sprintf() but replaces the argument list with _ap	7-99

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

Function	Description	Page
void <b>abort</b> (void)	Terminates a program abnormally	7-39
int <b>abs</b> (int j);	Returns the absolute value of j; expands inline unless <code>-x0</code> is used	7-40
void <b>atexit</b> (void (*fun)(void));	Registers the function pointed to by fun, called without arguments at normal program termination	7-44
double <b>atof</b> (const char *st);	Converts a string to a floating-point value; expands inline if <code>-x</code> is used	7-44
int <b>atoi</b> (register const char *st);	Converts a string to an integer value	7-44
long <b>atol</b> (register const char *st);	Converts a string to a long integer value; expands inline if <code>-x</code> is used	7-44
void <b>*bsearch</b> (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	7-45
void <b>*calloc</b> (size_t num, size_t size);	Allocates and clears memory for num objects, each of size bytes	7-46
div_t <b>div</b> (register int numer, register int denom);	Divides numer by denom producing a quotient and a remainder	7-50
void <b>exit</b> (int status);	Terminates a program normally	7-51
void <b>free</b> (void *packet);	Deallocates memory space allocated by malloc, calloc, or realloc	7-57
char <b>*getenv</b> (const char *_string)	Returns the environment information for the variable associated with _string	7-60
long <b>labs</b> (long i);	Returns the absolute value of i; expands inline unless <code>-x0</code> is used	7-40
long long <b>llabs</b> (long long i);	Returns the absolute value of i; expands inline	7-40
ldiv_t <b>ldiv</b> (long numer, long denom);	Divides numer by denom	7-63
int <b>ltoa</b> (long val, char *buffer);	Converts val to the equivalent string	7-65
void <b>*malloc</b> (size_t size);	Allocates memory for an object of size bytes	7-66
void <b>*memalign</b> (size_t alignment, size_t size);	Allocates memory for an object of size bytes aligned to an alignment byte boundary	7-66
void <b>minit</b> (void);	Resets all the memory previously allocated by malloc, calloc, or realloc	7-69

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

Function	Description	Page
void <b>qsort</b> (void *_base, size_t nmemb, size_t size, int (*compar) (void));	Sorts an array of nmemb members; <i>base</i> points to the first member of the unsorted array, and <i>size</i> specifies the size of each member	7-74
int <b>rand</b> (void);	Returns a sequence of pseudorandom integers in the range 0 to RAND_MAX	7-74
void <b>realloc</b> (void *packet, size_t size);	Changes the size of an allocated memory space	7-75
void <b>srand</b> (unsigned int seed);	Resets the random number generator	7-74
double <b>strtod</b> (const char *st, char **endptr);	Converts a string to a floating-point value	7-92
long <b>strtol</b> (const char *st, char **endptr, int base);	Converts a string to a long integer	7-92
long long <b>strtoll</b> (const char *st, char **endptr, int base);	Converts a string to a long long integer	7-92
unsigned long <b>strtoul</b> (const char *st, char **endptr, int base);	Converts a string to an unsigned long integer	7-92
unsigned long long <b>strtoull</b> (const char *st, char **endptr, int base);	Converts a string to an unsigned long long integer	7-92

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

Function	Description	Page
void <b>*memchr</b> (const void *cs, int c, size_t n);	Finds the first occurrence of <i>c</i> in the first <i>n</i> characters of <i>s</i> ; expands inline if <i>-x</i> is used	7-67
int <b>memcmp</b> (const void *cs, const void *ct, size_t n);	Compares the first <i>n</i> characters of <i>cs</i> to <i>ct</i> ; expands inline if <i>-x</i> is used	7-67
void <b>*memcpy</b> (void *s1, const void *s2, register size_t n);	Copies <i>n</i> characters from <i>s2</i> to <i>s1</i>	7-68
void <b>*memmove</b> (void *s1, const void *s2, size_t n);	Moves <i>n</i> characters from <i>s2</i> to <i>s1</i>	7-68
void <b>*memset</b> (void *mem, register int ch, register size_t length);	Copies the value of <i>ch</i> into the first <i>length</i> characters of <i>mem</i> ; expands inline if <i>-x</i> is used	7-68
char <b>*strcat</b> (char *string1, const char *string2);	Appends <i>string2</i> to the end of <i>string1</i>	7-81
char <b>*strchr</b> (const char *string, int c);	Finds the first occurrence of character <i>c</i> in <i>s</i> ; expands inline if <i>-x</i> is used	7-81
int <b>strcmp</b> (register const char *string1, register const char *s2);	Compares strings and returns one of the following values: <0 if <i>string1</i> is less than <i>string2</i> ; 0 if <i>string1</i> is equal to <i>string2</i> ; >0 if <i>string1</i> is greater than <i>string2</i> . Expands inline if <i>-x</i> is used.	7-82

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

Function	Description	Page
int <b>strcoll</b> (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	7-82
char * <b>strcpy</b> (register char *dest, register const char *src);	Copies string src into dest; expands inline if -x is used	7-83
size_t <b>strcspn</b> (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	7-84
char * <b>strerror</b> (int errno);	Maps the error number in errno to an error message string	7-84
size_t <b>strlen</b> (char *string);	Returns the length of a string	7-86
char * <b>strncat</b> (char *dest, const char *src, register size_t n);	Appends up to n characters from src to dest	7-87
int <b>strncmp</b> (const char *string1, const char *string2, size_t n);	Compares up to n characters in two strings; expands inline if -x is used	7-88
char * <b>strncpy</b> (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	7-89
char * <b>strpbrk</b> (const char *string, const char *chs);	Locates the first occurrence in string of any character from chs	7-90
char * <b>strrchr</b> (const char *string, int c);	Finds the last occurrence of character c in string; expands inline if -x is used	7-90
size_t <b>strspn</b> (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	7-91
char * <b>strstr</b> (register const char *string1, const char *string2);	Finds the first occurrence of string2 in string1	7-91
char * <b>strtok</b> (char *str1, const char *str2);	Breaks str1 into a series of tokens, each delimited by a character from str2	7-93
size_t <b>strxfrm</b> (register char *to, register const char *from, register size_t n);	Transforms n characters from <i>from</i> , to <i>to</i>	7-94

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

Function	Description	Page
char <b>*asctime</b> (const struct tm *timeptr);	Converts a time to a string	7-41
clock_t <b>clock</b> (void);	Determines the processor time used	7-47
char <b>*ctime</b> (const time_t *timer);	Converts calendar time to local time	7-49
double <b>difftime</b> (time_t time1, time_t time0);	Returns the difference between two calendar times	7-49
struct tm <b>*gmtime</b> (const time_t *timer);	Converts calendar time to Greenwich Mean Time	7-61
struct tm <b>*localtime</b> (const time_t *timer);	Converts calendar time to local time	7-64
time_t <b>mktime</b> (register struct tm *tptr);	Converts local time to calendar time	7-70
size_t <b>strftime</b> (char *out, size_t maxsize, const char *format, const struct tm *time);	Formats a time into a character string	7-85
time_t <b>time</b> (time_t *timer);	Returns the current calendar time	7-95



## 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 7-51.

**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 TMS470 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  $[0,\pi]$  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 7.3.17, *Time Functions (time.h/ctime)*, on page 7-27.

**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  $[-\pi/2, \pi/2]$  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  $[-\pi/2, \pi/2]$  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  $[-\pi, \pi]$  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 TMS470 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 `ptr1` and `ptr2` point to and returns one of the following values:

```
< 0   if *ptr1 is less than *ptr2
0     if *ptr1 is equal to *ptr2
> 0   if *ptr1 is greater than *ptr2
```

### calloc

### *Allocate and Clear Memory*

---

#### Syntax 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 linker with the `-heap` 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.

```
prrt = 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 7.3.17, *Time Functions (time.h/ctime)*, on page 7-27.

<b>cos</b>	<i>Cosine</i>
<b>Syntax for C</b>	<pre>#include &lt;math.h&gt;  double <b>cos</b>(double x);</pre>
<b>Syntax for C++</b>	<pre>#include &lt;cmath&gt;  double <b>std::cos</b>(double x);</pre>
<b>Defined in</b>	cos.c in rts.src
<b>Description</b>	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.
<b>Example</b>	<pre>double radians, cval; /* cos returns cval */ radians = 3.1415927; <b>cval = cos(radians);</b> /* return value = -1.0 */</pre>

<b>cosh</b>	<i>Hyperbolic Cosine</i>
<b>Syntax for C</b>	<pre>#include &lt;math.h&gt;  double <b>cosh</b>(double x);</pre>
<b>Syntax for C++</b>	<pre>#include &lt;cmath&gt;  double <b>std::cosh</b>(double x);</pre>
<b>Defined in</b>	cosh.c in rts.src
<b>Description</b>	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.
<b>Example</b>	<pre>double x, y;  x = 0.0; <b>y = cosh(x);</b> /* return value = 1.0 */</pre>

**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 7.3.17, *Time Functions (time.h/ctime)*, on page 7-27.

**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 7.3.17, *Time Functions (time.h/ctime)*, on page 7-27.

**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 TMS470 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 7-39.)

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 */
```

<b>fabs</b>	<i>Absolute Value</i>
<b>Syntax for C</b>	<pre>#include &lt;math.h&gt;  double <b>fabs</b>(double x);</pre>
<b>Syntax for C++</b>	<pre>#include &lt;cmath&gt;  double <b>std::fabs</b>(double x);</pre>
<b>Defined in</b>	fabs.c in rts.src
<b>Description</b>	The fabs function returns the absolute value of a floating-point number, x.
<b>Example</b>	<pre>double x, y;  x = -57.5; <b>y = fabs(x);</b>           /* return value = +57.5 */</pre>

<b>fclose</b>	<i>Close File</i>
<b>Syntax for C</b>	<pre>#include &lt;stdio.h&gt;  int <b>fclose</b>(FILE *_fp);</pre>
<b>Syntax for C++</b>	<pre>#include &lt;cstdio&gt;  int <b>std::fclose</b>(FILE *_fp);</pre>
<b>Defined in</b>	fclose.c in rts.src
<b>Description</b>	The fclose function flushes the stream that _fp points to and closes the file associated with that stream.

<b>feof</b>	<i>Test EOF Indicator</i>
<b>Syntax for C</b>	<pre>#include &lt;stdio.h&gt;  int <b>feof</b>(FILE *_fp);</pre>
<b>Syntax for C++</b>	<pre>#include &lt;cstdio&gt;  int <b>std::feof</b>(FILE *_fp);</pre>
<b>Defined in</b>	feof.c in rts.src
<b>Description</b>	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**

error.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_t fread(void *_ptr, size_t size, size_t count, FILE *_fp);
```

##### Syntax for C++

```
#include <cstdio>

size_t std::fread(void *_ptr, size_t size, size_t count, FILE *_fp);
```

##### Defined in

fread.c in rts.src

##### Description

The fread function reads from the stream pointed to by \_fp. The input is stored in the array pointed to by \_ptr. The number of objects read is \_count. The size of the objects is \_size.

<b>free</b>	<i>Deallocate Memory</i>
<b>Syntax for C</b>	<pre>#include &lt;stdlib.h&gt;  void <b>free</b>(void *ptr);</pre>
<b>Syntax for C++</b>	<pre>#include &lt;cstdlib&gt;  void <b>std::free</b>(void *ptr);</pre>
<b>Defined in</b>	memory.c in rts.src
<b>Description</b>	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, <i>Dynamic Memory Allocation</i> , on page 6-5.
<b>Example</b>	<p>This example allocates ten bytes and then frees them.</p> <pre>char *x; x = malloc(10);          /* allocate 10 bytes */ <b>free</b>(x);                /* free 10 bytes */</pre>
<b>freopen</b>	<i>Open File</i>
<b>Syntax for C</b>	<pre>#include &lt;stdio.h&gt;  FILE *<b>freopen</b>(const char *_fname, const char *_mode, register FILE *_fp);</pre>
<b>Syntax for C++</b>	<pre>#include &lt;cstdio&gt;  FILE *<b>std::freopen</b>(const char *_fname, const char *_mode,                     register FILE *_fp);</pre>
<b>Defined in</b>	freopen.c in rts.src
<b>Description</b>	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  $\text{value} = x \times 2^{\text{exp}}$ . 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 7.3.17, *Time Functions (time.h/ctime)*, on page 7-27.

**isxxx***Character Typing*

---

**Syntax for C**

#include &lt;ctype.h&gt;

int <b>isalnum</b> (int c);	int <b>islower</b> (int c);
int <b>isalpha</b> (int c);	int <b>isprint</b> (int c);
int <b>isascii</b> (int c);	int <b>ispunct</b> (int c);
int <b>iscntrl</b> (int c);	int <b>isspace</b> (int c);
int <b>isdigit</b> (int c);	int <b>isupper</b> (int c);
int <b>isgraph</b> (int c);	int <b>isxdigit</b> (int c);

**Syntax for C++**

#include &lt;cctype&gt;

int <b>std::isalnum</b> (int c);	int <b>std::islower</b> (int c);
int <b>std::isalpha</b> (int c);	int <b>std::isprint</b> (int c);
int <b>std::isascii</b> (int c);	int <b>std::ispunct</b> (int c);
int <b>std::iscntrl</b> (int c);	int <b>std::isspace</b> (int c);
int <b>std::isdigit</b> (int c);	int <b>std::isupper</b> (int c);
int <b>std::isgraph</b> (int c);	int <b>std::isxdigit</b> (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:

<b>isalnum</b>	Identifies alphanumeric ASCII characters (tests for any character for which <b>isalpha</b> or <b>isdigit</b> is true)
<b>isalpha</b>	Identifies alphabetic ASCII characters (tests for any character for which <b>islower</b> or <b>isupper</b> is true)
<b>isascii</b>	Identifies ASCII characters (any character from 0–127)
<b>iscntrl</b>	Identifies control characters (ASCII characters 0–31 and 127)
<b>isdigit</b>	Identifies numeric characters between 0 and 9 (inclusive)
<b>isgraph</b>	Identifies any nonspace character
<b>islower</b>	Identifies lowercase alphabetic ASCII characters
<b>isprint</b>	Identifies printable ASCII characters, including spaces (ASCII characters 32–126)
<b>ispunct</b>	Identifies ASCII punctuation characters
<b>isspace</b>	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 7-40.*

---

**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  $2^{\text{exp}}$  and returns  $x \times 2^{\text{exp}}$ . 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 7-50.*

---

localtime	<i>Local Time</i>
<b>Syntax for C</b>	<pre>#include &lt;time.h&gt;  <b>struct tm *localtime</b>(const time_t *timer);</pre>
<b>Syntax for C++</b>	<pre>#include &lt;ctime&gt;  <b>struct tm *std::localtime</b>(const time_t *timer);</pre>
<b>Defined in</b>	localtime.c in rts.src
<b>Description</b>	<p>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.</p> <p>For more information about the functions and types that the time.h/ctime header declares and defines, see section 7.3.17, <i>Time Functions (time.h/ctime)</i>, on page 7-27.</p>

log	<i>Natural Logarithm</i>
<b>Syntax for C</b>	<pre>#include &lt;math.h&gt;  double <b>log</b>(double x);</pre>
<b>Syntax for C++</b>	<pre>#include &lt;cmath&gt;  double <b>std::log</b>(double x);</pre>
<b>Defined in</b>	log.c in rts.src
<b>Description</b>	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.
<b>Description</b>	<pre>float x, y;  x = 2.718282; <b>y = log(x);</b>           /* Return value = 1.0 */</pre>

**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 7-77.*

---

**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 linker with the `-heap` 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 `__SYSTEM_SIZE` defines the size of the heap as 2K bytes. You can change this amount at link time by invoking the linker with the `-heap` option and specifying the desired size of the heap (in bytes) directly after the option. 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 7.3.17, *Time Functions (time.h/ctime)*, on page 7-27.

#### 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  $y \leq 0$ , or if x is negative and y is not an integer. A range error occurs if the result is too large to represent.

**Example**

```
double x, y, z;

x = 2.0;
y = 3.0;
x = pow(x, y);           /* return value = 8.0 */
```

**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.

<b>putchar</b>	<i>Write Character to Standard Output</i>
<b>Syntax for C</b>	<pre>#include &lt;stdlib.h&gt;  int <b>putchar</b>(int _x);</pre>
<b>Syntax for C++</b>	<pre>#include &lt;cstdlib&gt;  int <b>std::putchar</b>(int _x);</pre>
<b>Defined in</b>	putchar.c in rts.src
<b>Description</b>	The putchar function writes a character to the standard output device.

<b>puts</b>	<i>Write to Standard Output</i>
<b>Syntax for C</b>	<pre>#include &lt;stdlib.h&gt;  int <b>puts</b>(const char *_ptr);</pre>
<b>Syntax for C++</b>	<pre>#include &lt;cstdlib&gt;  int <b>std::puts</b>(const char *_ptr);</pre>
<b>Defined in</b>	puts.c in rts.src
<b>Description</b>	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 `__SYSTEM_SIZE` defines the size of the heap as 2K bytes. You can change this amount at link time by invoking the linker with the `-heap` option and specifying the desired size of the heap (in bytes) directly after the option. 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

#### *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_buf env)
void longjmp(jmp_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 7-74.*

---

**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.

**strptime***Format Time***Syntax for C**

```
#include <time.h>
```

```
size_t *strptime(char *s, size_t maxsize, const char *format,
                  const struct tm *timeptr);
```

**Syntax for C++**

```
#include <ctime>
```

```
size_t *std::strptime(char *s, size_t maxsize, const char *format,
                      const struct tm *timeptr);
```

**Defined in**

strptime.c in rts.src

**Description**

The `strptime` 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 `strptime` 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 <i>weekday</i> name (Mon, Tue, . . . )
%A	The full <i>weekday</i> name
%b	The abbreviated <i>month</i> name (Jan, Feb, . . . )
%B	The locale's full <i>month</i> name
%c	The <i>date</i> and <i>time</i> representation
%d	The <i>day</i> of the month as a decimal number (0–31)
%H	The <i>hour</i> (24-hour clock) as a decimal number (00–23)
%I	The <i>hour</i> (12-hour clock) as a decimal number (01–12)
%j	The <i>day</i> of the year as a decimal number (001–366)
%m	The <i>month</i> as a decimal number (01–12)
%M	The <i>minute</i> as a decimal number (00–59)
%p	The locale's equivalency of either <i>a.m.</i> or <i>p.m.</i>
%S	The <i>seconds</i> as a decimal number (00–50)
%U	The <i>week</i> number of the year (Sunday is the first day of the week) as a decimal number (00–52)
%x	The <i>date</i> representation
%X	The <i>time</i> representation
%y	The <i>year</i> without century as a decimal number (00–99)

Character	Expands to
%Y	The <i>year</i> with century as a decimal number
%Z	The <i>time zone</i> 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 7.3.17, *Time Functions (time.h/ctime)*, on page 7-27.

## 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 `src` 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 *src = "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" */;
/* src--> "I am the one father warned you of\0" */;
/* strd--> "oops\0" */;

strncpy (stra, src, length);

/* stra--> "I am the one mother warned you of\0" */;
/* strb--> " he's\0" */;
/* src--> "I am the one father warned you of\0" */;
/* strd--> "oops\0" */;

strncpy (stra, strd, length);

/* stra--> "oops\0" */;
/* strb--> " he's\0" */;
/* src--> "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 std::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  $\pm\text{HUGE\_VAL}$ ; 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 */
```



<b>time</b>	<i>Time</i>
<b>Syntax for C</b>	<pre>#include &lt;time.h&gt;  time_t <b>time</b>(time_t *timer);</pre>
<b>Syntax for C++</b>	<pre>#include &lt;ctime&gt;  time_t <b>std::time</b>(time_t *timer);</pre>
<b>Defined in</b>	time.c in rts.src
<b>Description</b>	<p>The time function determines the current calendar time, represented in seconds. If the calendar time is not available, the function returns <code>-1</code>. If timer is not a null pointer, the function also assigns the return value to the object that timer points to.</p> <p>For more information about the functions and types that the time.h/ctime header declares and defines, see section 7.3.17, <i>Time Functions (time.h/ctime)</i>, on page 7-27.</p> <div> <p><b>Note: The time Function Is Target-System Specific</b></p> <p>The time function is target-system specific, so you must write your own time function.</p> </div>
<b>tmpfile</b>	<i>Create Temporary File</i>
<b>Syntax for C</b>	<pre>#include &lt;stdlib.h&gt;  FILE *<b>tmpfile</b>(void);</pre>
<b>Syntax for C++</b>	<pre>#include &lt;cstdlib&gt;  FILE *<b>std::tmpfile</b>(void);</pre>
<b>Defined in</b>	tmpfile.c in rts.src
<b>Description</b>	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                      */
}
```

## fprintf

### *Write to Stream*

---

#### Syntax for C

```
#include <stdlib.h>

int fprintf(FILE *_fp, const char *_format, va list _ap);
```

#### Syntax for C++

```
#include <cstdlib>

int std::fprintf(FILE *_fp, const char *_format, va list _ap);
```

#### 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. The argument list is given by `_ap`.

## fprintf

### *Write to Standard Output*

---

#### Syntax for C

```
#include <stdlib.h>

int fprintf(const char *_format, va list _ap);
```

#### Syntax for C++

```
#include <cstdlib>

int std::fprintf(const char *_format, va list _ap);
```

#### Defined in

fprintf.c in rts.src

#### Description

The `fprintf` 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`.

---

# Library-Build Utility

When using the TMS470R1x 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 the source archive, `rts.src`, which contain all run-time-support functions:

The following prebuilt run-time-support libraries are included with the package:

- ☐ `rts16.lib` (16-bit big-endian processing)
- ☐ `rts16e.lib` (16-bit little-endian processing)
- ☐ `rts32.lib` (32-bit big-endian processing)
- ☐ `rts32e.lib` (32-bit little-endian processing)

You can also build your own run-time-support libraries by using the `mk470` utility described in this chapter and the archiver described in the *TMS470R1x Assembly Language Tools User's Guide*.

Topic	Page
8.1 Invoking the Library-Build Utility .....	8-2
8.2 Library-Build Utility Options .....	8-3
8.3 Options Summary .....	8-4

## 8.1 Invoking the Library-Build Utility

The syntax for invoking the library-build utility is:

```
mk470 [options] src_arch1 [-lobj.lib1] [src_arch2 [-lobj.lib2]] ...
```

- mk470**      Command that invokes the utility.
- options*      Options affect how the library-build utility treats your files. Options can appear anywhere on the command line or in a linker command file. (Options are discussed in section 8.2 and section 8.3.)
- src\_arch*    The name of a source archive file. For each source archive named, mk470 builds an object library according to the run-time model specified by the command-line options.
- lobj.lib**   The optional object library name. If you do not specify a name for the library, mk470 uses the name of the source archive and appends a *.lib* suffix. For each source archive file specified, a corresponding object library file is created. You cannot build an object library from multiple source archive files.

The mk470 utility runs the compiler on each source file in the archive to compile and/or assemble it. Then, the utility collects all the object files into the object library. All the tools must be in your PATH environment variable. The utility ignores the environment variables TMP, C\_OPTION, and C\_DIR.



## 8.2 Library-Build Utility Options

Most of the options that are included on the command line correspond directly to options of the same name used with the compiler, assembler, linker, and shell. The following options apply only to the library-build utility.

- c** Extracts C source files contained in the source archive from the library and leaves them in the current directory after the utility completes execution.
- h** Uses header files contained in the source archive and leaves them in the current directory after the utility completes execution. Use this option to install the run-time-support header files from the `rtsc.src` or `rtscpp.src` archive that is shipped with the tools.
- k** Overwrites files. By default, the utility aborts any time it attempts to create an object file when an object file of the same name already exists in the current directory, regardless of whether you specified the name or the utility derived it.
- q** Suppresses header information (quiet).
- u** Does not use the header files contained in the source archive when building the object library. If the desired headers are already in the current directory, there is no reason to reinstall them. This option gives you flexibility in modifying run-time-support functions to suit your application.
- v** Prints progress information to the screen during execution of the utility. Normally, the utility operates silently (no screen messages).

For an online summary of the options, enter **mk470** with no parameters on the command line.

## 8.3 Options Summary

The other options you can use with the library-build utility correspond directly to the options used with the compiler and assembler. Table 8–1 lists these options, and they are described in detail on the page indicated in the table.

*Table 8–1. Summary of Options and Their Effects*

*(a) Options that control the compiler/shell*

Option	Effect	Page
<code>-Dname[=def]</code>	Predefines <i>name</i>	2-15
<code>-g</code> or <code>--symdebug:dwarf</code>	Enables symbolic debugging, using the DWARF2 debug format	2-18, 3-16
<code>--symdebug:coff</code>	Enables symbolic debugging using the alternate STABS debugging format	2-19, 3-16
<code>-k</code>	Keeps .asm file	2-15
<code>-Uname</code>	Undefines <i>name</i>	2-17

*(b) Options that control the parser*

Option	Effect	Page
<code>-pe</code>	Enables embedded C++ mode	5-28
<code>-pi</code>	Disables definition-controlled inlining (but <code>-o3</code> optimizations still perform automatic inlining)	2-39
<code>-pk</code>	Makes code K&R compatible	5-26
<code>-pr</code>	Enables relaxed mode; ignores strict ANSI violations	5-28
<code>-ps</code>	Enables strict ANSI mode (for C/C++, not K&R C)	5-28

*(c) Parser options that control diagnostics*

Option	Effect	Page
<code>-pdr</code>	Issues remarks (nonserious warnings)	2-32
<code>-pdv</code>	Provides verbose diagnostics that display the original source with line wrap	2-33
<code>-pdw</code>	Suppresses warning diagnostics (errors are still issued)	2-33

*(d) Options that control the optimization level*

Option	Effect	Page
-O0	Optimizes register usage	3-2
-O1	Uses -O0 optimizations and optimizes locally	3-2
-O2 or -O	Uses -O1 optimizations and optimizes globally	3-3
-O3	Uses -O2 optimizations and optimizes file	3-3, 3-4

*Table 8–1. Summary of Options and Their Effects (Continued)**(e) Options that change the C run-time model*

Option	Effect	Page
-ma	Assumes variables are aliased	2-16
-mc	Changes variables of type char from unsigned to signed	2-16
-md	Disables dual-state interworking support	6-43
-me	Produces code for little-endian format	2-16
-mf	Optimizes for speed over space	2-16
-ml	Emits Long calls	6-19
-mn	Enables optimizer options disabled by -g	2-16
-mt	Generates 16-bit code	2-16

*(f) Option that controls the assembler*

Option	Effect	Page
-as	Keeps labels as symbols	2-22

*(g) Options that change the default file extensions*

Option	Effect	Page
-ea[.] <i>extension</i>	Sets default extension for assembly files	2-20
-eo[.] <i>extension</i>	Sets default extension for object files	2-20

---

# C++ Name Demangler

---

---

---

The C++ compiler implements function overloading, operator overloading, and type-safe linking by encoding a function's signature in its link-level name. The process of encoding the signature into the linkname is often referred to as *name mangling*. When you inspect mangled names, such as in assembly files or linker output, it can be difficult to associate a mangled name with its corresponding name in the C++ source code. The C++ *name demangler* is a debugging aid that translates each mangled name it detects to its original name found in the C++ source code.

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 .....	9-2
9.2 C++ Name Demangler Options .....	9-2
9.3 Sample Usage of the C++ Name Demangler .....	9-3

## 9.1 Invoking the C++ Name Demangler

To invoke the C++ name demangler, enter:

**dem470** [*options*] [*filenames*]

**dem470**    Command that runs the name demangler

*options*    Options affect how the demangler behaves. Options can appear anywhere on the command line or in a linker command file. (Options are discussed in section 9.2.)

*filenames*    Text input files, such as the assembly file output by the compiler, the assembler listing file, and the linker map file. If no filenames are specified on the command line, dem470 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.

<b>-h</b>	Prints a help screen that provides an online summary of the C++ name demangler options
<b>-o file</b>	Outputs to the given <i>file</i> rather than to standard out
<b>-u</b>	External names do not have a C++ prefix
<b>-v</b>	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 TMS470 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:
;* -----*
    STMFD    SP!, {A3, A4, LR}
    ADD      A1, SP, #4           ; |15|
    BL       ____ct__6bananaFv   ; |15|
; |15|
    ADD      A1, SP, #4           ; |16|
    BL       _calories__6bananaFv ; |16|
; |16|
    STR      A1, [SP, #0]         ; |16|
    ADD      A1, SP, #4           ; |16|
    MOV      A2, #2              ; |16|
    BL       ____dt__6bananaFv   ; |16|
; |16|
    LDR      A1, [SP, #0]         ; |16|
    LDMFD    SP!, {A3, A4, PC}
```

Executing the C++ name demangler utility demangles all names that it believes to be mangled. If you enter:

% **dem470 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():
;* -----*
    STMFD    SP!, {A3, A4, LR}
    ADD      A1, SP, #4           ; |15|
    BL       banana::banana()    ; |15|
; |15|
    ADD      A1, SP, #4           ; |16|
    BL       banana::_calories() ; |16|
; |16|
    STR      A1, [SP, #0]         ; |16|
    ADD      A1, SP, #4           ; |16|
    MOV      A2, #2              ; |16|
    BL       banana::~~banana()  ; |16|
; |16|
    LDR      A1, [SP, #0]         ; |16|
    LDMFD    SP!, {A3, A4, PC}
```



# Static Stack Depth Profiler

The static stack depth profiler provides information about the maximum stack depth requirements of an application based on the static information available in the linker output file. The application must be compiled with DWARF (default) type debug information.

The profiler is a stand-alone application called sdp470. The profiler takes a linked output file as input and produces a listing that details the stack usage of all of the functions defined in the application. If an application contains indirect calls and/or reentrant procedures, then a configuration file should also be provided as input to the profiler.

Topic	Page
10.1 Invoking the Static Stack Depth Profiler .....	10-2
10.2 Stack Depth Statistics Listing .....	10-3
10.3 Dependencies and Limitations .....	10-5
10.4 User Input Mechanisms .....	10-6
10.5 Configuration File Specification .....	10-7
10.6 Profiler Generated Warnings .....	10-12
10.7 Run-Time Support for Stack Depth Profiling .....	10-13

## 10.1 Invoking the Static Stack Depth Profiler

The syntax for invoking the static stack depth profiler is as follows:

**sdp470** [-c *config*] *outfile*

<b>sdp470</b>	Command that runs the compiler and the assembler
<b>-c <i>config</i></b>	Identifies a configuration file to be used by the profiler to supply information about indirectly called functions and reentrant procedures. For more information, see section 10.5, <i>Configuration File Specification</i> , on page 10-7.
<b><i>outfile</i></b>	Identifies the linked output file for an application to be analyzed by the profiler. This file contains debug information about all functions included in the final link of an application.

The entry point, or root function, is identified in the linked output file. The profiler uses the same method to determine the entry point as the linker:

- 1) A global symbol identified with -e linker command option
- 2) The value of the `_c_int00` symbol (if present)
- 3) The value of the `_main` or `$main` symbol (if present)
- 4) Address 0x0 (the default)

## 10.2 Stack Depth Statistics Listing

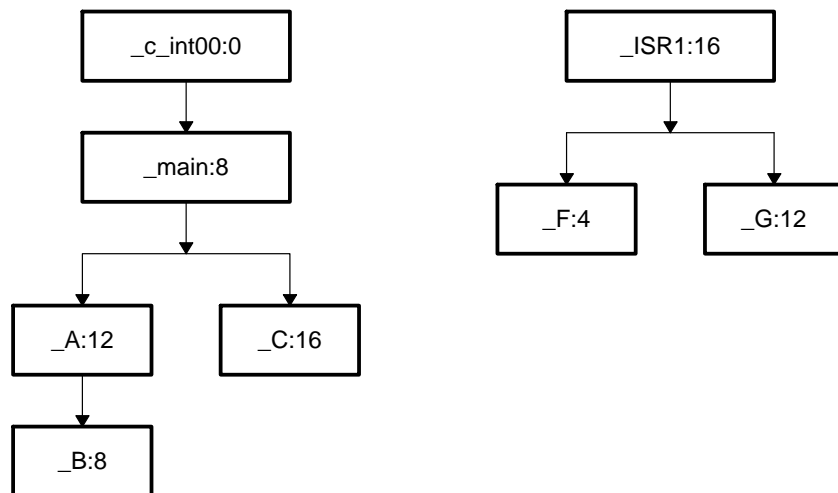
Each segment of the listing details the stack usage for a particular function in the application. Function segments are listed in order, beginning with the function that has the highest total stack need.

The first line in a segment provides details about a given parent function. The first line in a function segment provides information on the stack space needed by the function and its total stack usage estimate (the function's stack usage plus the worst stack usage of its callees).

Subsequent lines in a function segment list the callees of the parent function listed in the first line of the segment. The callees are listed in order from highest to lowest stack usage.

Figure 3–1 illustrates the application call trees for an example application:

*Figure 3–1. Application Call Trees*



Example 10–1 shows the function segment listings for the application illustrated in Figure 3–1.

*Example 10–1. Profile of the Application Call Trees Figure*

```

*****
* Static Stack Depth Analysis Profile
*****
FCN:cint00      stack usage: 0, total stack need:    28
  _main        stack usage: 8, subtree stack need:   28
=====
FCN:_main      stack usage: 8, total stack need:    28
  _A          stack usage: 12, subtree stack need:   20
  _C          stack usage: 16, subtree stack need:   16
=====
FCN:_ISR1      stack usage: 16, total stack need:    28
  _G          stack usage: 12, subtree stack need:   12
  _F          stack usage: 4, subtree stack need:    4
=====
FCN:_A         stack usage: 12, total stack need:    20
  _B          stack usage: 8, subtree stack need:    8
=====
FCN:_C         stack usage: 16, total stack need:    16
=====
FCN:_G         stack usage: 12, total stack need:    12
=====
FCN:_B         stack usage: 8, total stack need:     8
=====
FCN:_F         stack usage: 4, total stack need:     4
=====

```

A summary of the application's stack depth requirements is provided at the end of the listing. The summary states an estimate of the stack depth usage from each function in the application that does not have a parent. You can derive a worst case estimate of an application's stack depth using the information provided in the listing in combination with your knowledge of which functions are interrupts (and which stack mode those interrupts assume).

For example, a summary of Example 10–1 looks like this:

```

=====ROOT FUNCTIONS=====
_c_int00      total stack need:    28 bytes
_ISR1        total stack need:    28 bytes

```

## 10.3 Dependencies and Limitations

The profiler constructs a call tree based on the DWARF debug information provided in the linker output file. The tree is annotated with the stack usage information that the profiler derives from the function's debug information.

There are several significant limitations to the profiler's ability to complete this information:

- ☐ Hand-coded assembly functions and stack usage information for those functions
- ☐ Indirectly called functions and their callers
- ☐ Reentrant functions and their depth of recursion

The profiler relies on your input to supply this information in the application source code and in the configuration file (specified with the `-c` option). The assembler processes the assembly source code annotations (`.asmfunc/.endasmfunc` directives) and encodes the information into the linked output file in a form that the profiler can understand. You must also provide a configuration file to identify indirect calls and reentrant procedures to the profiler. The profiler uses this configuration file to annotate the call graph that it constructed from the linked output file for the application.


The accuracy of the stack depth estimate is heavily dependent on the accuracy of the information you provide. You must actively maintain this information throughout the life of a given application.

## 10.4 User Input Mechanisms

The assembler supports the `.asmfunc` and `.endasmfunc` assembler directives for identifying the beginning and end of an assembly function. The `.asmfunc` directive has an optional parameter for specifying the stack space needed for an assembly function.

For example, you could write a function, `$foo()`, which uses 12 bytes of stack space, as shown in Example 10–2:

### *Example 10–2. An Assembly Function*

```
.global    $foo
$foo: .asmfunc    stack_usage(12)
      PUSH      { R4, R5, R6 }
      
      POP       { R4, R5, R6 }
      MOV       PC, LR
      .endasmfunc
```

The `.asmfunc` and `.endasmfunc` directives identify the entry and exit points of the function. This information and the fact that `$foo` uses 12 bytes of stack space are then encoded into the debug information. Assembler functions must be annotated with these directives to be considered for stack depth analysis and profile report.

## 10.5 Configuration File Specification

The stack depth profiler enables you to specify indirect calls and reentrant procedures using a configuration file. The configuration file is input to the profiler, along with the executable to be analyzed. The profiler applies the configuration information to the stack depth information extracted from the executable and reports the total stack usage.

### 10.5.1 Specify Indirect Calls

The configuration file is composed of a series of sections delimited by the section type. Each section contains information for a specific section type. For example, the section specifying indirectly called functions is delimited by `<INDIRECT>` and `</INDIRECT>`, and contains information on the set of procedures called indirectly.

#### *Example 10–3. Specifying Indirect Calls*

```
<INDIRECT>
func1: callee1[(mode)], callee2[(mode)],2, calleeN[(mode)]
func2: callee1[(mode)],2, calleeN[(mode)]
2
funcm: callee1[(mode)],2, calleeN[(mode)]
</INDIRECT>
```

The keywords `<INDIRECT>` and `</INDIRECT>` delimit an indirect callee section. Each line in the section begins with the name of a calling procedure, `func`, followed by a colon (:), which is then followed by a list of functions that are called indirectly from `func`. Each entry in the indirect callee list consists of the callee name, and an optional mode (in parentheses) in which the callee was compiled.

The profiler recognizes the specification of one of three modes: ARM, THUMB, and DUAL (default). If a mode is not specified, the profiler will assume DUAL mode. For DUAL mode callee functions, the profiler automatically adds the names of the callees (in ARM and THUMB modes) to the callee list for `func`. In this example, function `bar` is specified as a callee in DUAL mode, and both `_bar` and `$bar` are added to the callee list for `main`:

```
<INDIRECT>
main:foo(ARM),bar
</INDIRECT>
```

Each indirect call can be specified on a separate line within an indirect call section:

```
<INDIRECT>
main:foo(ARM)
main: bar
</INDIRECT>
```

The profiler performs no semantic checks on the information input via the configuration file. That is, it does not check to determine if the procedures listed in the callee list or the caller list exist or if they exist in the mode specified. The profiler checks, however, for available debug information for a function and generates a warning if none is found.

The profiler declares the following types of loads of PC (in ARM) mode as indirect call points, where #imm is an immediate value.

```
LDR{B,H} PC, [reg, #imm]
```

However, loads of PC with register offsets are not identified as call points since such loads are used generally for switch table jumps.

## 10.5.2 Specifying Reentrant Procedures

Reentrant functions present a difficult problem to the profiler's ability to accurately estimate the worst-case space requirement of an application. One difficulty with reentrant functions is that the depth of recursion is often data-dependent. You may not be able to accurately inform the profiler about the depth of recursion without knowing in advance all of the input data sets that will be used in the application. Another difficulty is the presence of multiple recursive functions in a call-graph, which would make it hard to determine the recursive behavior of each function in the call-graph.

In light of these difficulties, no automated analysis is performed to determine the presence of reentrant functions or their nature. Instead, we must input the maximum recursive depth of a function. The profiler assumes that your input is correct and reliable. It issues a warning and warns if there is no apparent recursion in the call graph for a function and recursion depth that has been specified. Beyond the required analysis to issue such a warning, the profiler does not perform any analysis on reentrant calls.

The configuration file issued to specify the list of reentrant procedures in the application. The format is as follows:

```
<REENTRANT>  
func1 (depth)  
func2 (depth)  
</REENTRANT>
```

The reentrant section is delimited by the <REENTRANT> and </REENTRANT> keywords as shown above. Each line in the section consists of a function name followed by the recursion depth (in parentheses) to be assumed for the function. The depth value should be a valid, non-negative integer.



The profiler multiplies the stack size of the function by its specified depth to obtain the final stack usage requirement for the function. For example,

```
<REENTRANT>
foo (5)
</REENTRANT>
```

For this example, the profiler determines that the static stack usage requirement of foo is 100, the maximum dynamic requirement of foo would be  $100 \times 5 = 500$ . The depth of a function defaults to 1 in the absence of a depth field

The profiler does not make any effort to determine if the procedure specified in the reentrant section is indeed recursive. It assumes that the input is correct and reliable.

### 10.5.3 Specifying Interrupt Service Routines

Interrupt service routines (ISRs), are used to handle interrupts. An interrupt service routine is a special root function that is usually not directly called from the application. In ARM, an interrupt request can be serviced by an ISR in any of the six modes. Each mode typically uses a different stack area for the interrupt handler. The SDA can output the stack requirements of an ISR if you identify ISRs and their modes of operations. The syntax for specifying an ISR and its mode is given below

```
<INTERRUPT>
routine_name(mode)
</INTERRUPT>
```

Where routine\_name is the name of the ISR and mode can be one of the following:

- 1) irq (general purpose interrupt),
- 2) fiq (data transfer or channel process)
- 3) svc (operating system protected mode)
- 4) abt (data or instruction prefetch abort)
- 5) sys (privileged user mode for the operating system)
- 6) und (undefined instruction).

The stack space requirement for ISRs is output after the stack space requirements for routines directly called by the application.

### 10.5.4 Other Features

The configuration file supports all preprocessing directives supported by cl470. The configuration file should be processed using the cl470 preprocessor before being input to the profiler. For example, if file config1 contains these lines these lines:

```
<INDIRECT>
main:foo
</INDIRECT>
#include "config2"
```

and file "config2" contains these lines:

```
<REENTRANT>
foo(5)
</REENTRANT>
```

The cl470 preprocessor can be used to combine the two config files. For example:

```
cl470 -ppo config1
```

The preprocessor adds or replaces the file suffix with .pp. In the above example, the combined config files will be found in the file `config1.pp`.

The profiler supports C++ style comments. All characters following the `//` symbol to the end of the line (new line character) are ignored. The `//` can appear anywhere in a configuration file. For example

```
<INDIRECT>
// This is a comment
main:foo
// End of indirect callee list
</INDIRECT>
```

The configuration file does not permit nesting of sections. For example, the following specification is illegal

```
<INDIRECT>
foo:bar
<REENTRANT>
foo(5)
</REENTRANT>
</INDIRECT>
```

The profiler performs no semantic checks on the information specified in the configuration file and assumes that the information provided is correct and valid. It is your responsibility to ensure the validity of information specified in the configuration file.

### 10.5.5 Tail Calls

A tail call is code that branches to a routine `foo()` at the end of routine `bar()` so that control returns directly to `bar`'s caller at the end of `foo()`. The SDA handles tail calls by identifying tail call points as both call point and return points. The pseudo assembly instruction `CRET` is used for representing tail calls.

## 10.6 Profiler Generated Warnings

The static stack depth profiler detects the following situations and issues a warning in each case:

- ☐ A function contains an indirect branch (branch to the contents of a register) and there are no <INDIRECT>entries for the function in the configuration file, the profiler issues a warning that a potential indirect call has been observed and this caller/callee pair is not specified in the configuration file. Not all indirect branches are function calls; e.g., in 32-BIS mode, long branches are achieved through indirect branches.
- ☐ A set of indirect callees for a function is not specified and the profiler observes no indirect branches in the function, it issues a warning indicating the absence of indirect calls.
- ☐ A function's call graph has a link back to the function indicating potential recursion and no recursion depth is specified, the profiler issues a warning indicating that the function is reentrant. A warning and not an error is issued since a function could have an apparent recursion and not a real one.
- ☐ The profiler discovers no links back to the function in a call-graph and a recursion depth is specified, it issues a warning.
- ☐ No stack usage information for a function is found.
- ☐ No debug information for a function is found.
- ☐ The profiler generates a potential indirect callee warning for each routine compiled in dual mode. This is because, in dual mode, each call to an external routine is performed by jumping to the RTS routine IND\_CALL or IND\$CALL (depending on the mode of the caller)
- ☐ The profiler generates potential indirect callee warnings for routines in the RTS library.

## **10.7 Run-Time Support for Stack Depth Profiling**

The TMS470 compiler provides a debug option to determine the maximum stack usage of an application. If the application is compiled with the `-debug:sdp` option, the compiler places a call to a stack depth bookkeeping routine (`C_SDPBK` for 32-BIS and `C$SDPBK` for 16-BIS), immediately after the frame has been allocated for a function. This bookkeeping routine tracks the maximum stack usage for each function. There is a profiled version of the run-time support library that can be linked when `-debug:sdp` is turned on. This enables you to determine the complete stack usage of an application that contains run-time support utilities (such as I/O utilities).

There is an overhead associated with the `-debug:sdp` option. The link register (LR) is always saved when this option is turned on, since all functions in the application call the bookkeeping routine.

---

# Glossary

---

---

---

## A

**ANSI:** *American National Standards Institute.* An organization that establishes standards voluntarily followed by industries.

**alias disambiguation:** A technique that determines when two pointer expressions cannot point to the same location, allowing the compiler to freely optimize such expressions.

**aliasing:** Aliasing occurs when a single object can be accessed in more than one way, such as when two pointers point to a single object. It can disrupt optimization because any indirect reference could refer to any other object.

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

**archive library:** A collection of individual files grouped into a single file by the archiver.

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

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

**assignment statement:** A statement that initializes a variable with a value.

**autoinitialization:** The process of initializing global C/C++ variables (contained in the `.cinit` section) before program execution begins.

**autoinitialization at load time:** An autoinitialization method used by the linker when linking C/C++ code. The linker uses this method when you invoke the linker with the `-cr` option. This method initializes variables at load time instead of run time.

**autoinitialization at run time:** An autoinitialization method used by the linker when linking C/C++ code. The linker uses this method when you invoke the linker with the `-c` option. The linker loads the `.cinit` section of data tables into memory, and variables are initialized at 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 linker or hex conversion utility options and names input files for the linker or hex conversion utility.

**comment:** A source statement (or portion of a source statement) that documents or improves readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.

**common object file format (COFF):** A binary object file format that promotes modular programming by supporting the concept of *sections*. All COFF sections are independently relocatable in memory space; you can place any section into any allocated block of target memory.

**constant:** A type whose value cannot change.



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

## D

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

**direct call:** A function call where one function calls another using the function's name.

**directives:** Special-purpose commands that control the actions and functions of a software tool.

**disambiguation:** See *alias disambiguation*

**dynamic memory allocation:** A technique used by several functions (such as malloc, calloc, and realloc) to dynamically allocate memory for variables at 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 TMS470 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 utility:** 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.

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

**listing file:** An output file created by the assembler that lists source statements, their line numbers, and their effects on the section program counter (SPC).

**little endian:** An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *big endian*

**loader:** A device that loads an executable module into system memory.

**loop unrolling:** An optimization that expands small loops so that each iteration of the loop appears in your code. Although loop unrolling increases code size, it can improve the efficiency of your code.

**M**

**macro:** A user-defined routine that can be used as an instruction.

**macro call:** The process of invoking a macro.

**macro definition:** A block of source statements that define the name and the code that make up a macro.

**macro expansion:** The process of inserting source statements into your code in place of a macro call.

**map file:** An output file, created by the linker, that shows the memory configuration, section composition, section allocation, symbol definitions, and the addresses at which the symbols were defined for your program.

**memory map:** A map of target system memory space that is partitioned into functional blocks.

## O

**object file:** An assembled or linked file that contains machine-language object code.

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

**operand:** An argument of an assembly language instruction, assembler directive, or macro directive that supplies information to the operation performed by the instruction or directive.

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

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

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

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

## P

**parser:** A software tool that reads the source file, performs preprocessing functions, checks the syntax, and produces an intermediate file that can be used as input for the optimizer or code generator.

**pragma:** Preprocessor directive that provides directions to the compiler about how to treat a particular statement.

**preprocessor:** A software tool that interprets macro definitions, expands macros, interprets header files, interprets conditional compilation, and acts upon preprocessor directives.

**program-level optimization:** An aggressive level of optimization where all of the source files are compiled into one intermediate file. Because the compiler can see the entire program, several optimizations are performed with program-level optimization that are rarely applied during file-level optimization.

**R**

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

**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.

**shell program:** A utility that lets you compile, assemble, and optionally link in one step. The shell runs one or more source modules through the compiler (including the parser, optimizer, and code generator), the assembler, and the linker.

**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

---

---

---

% in time format 7-85  
-@ compiler option 2-15  
<> for shell input 2-27  
>> symbol 2-34  
0, removing comparisons to 3-29  
\_\_16bis\_\_ predefined name 2-26  
\_\_32bis\_\_ predefined name 2-26  
470cio.h header 7-2

## A

-a linker option 4-5  
-aa assembler option 2-22  
abort function 7-39  
abs function 2-38, 7-40  
absolute compiler limits 5-29  
absolute lister 1-4  
absolute listing, creating 2-22  
absolute value function 7-40, 7-52  
-ac assembler option 2-22  
acos function 7-40  
-ad assembler option 2-22  
add\_device function 7-7  
-ahc assembler option 2-22  
-al assembler option 2-22  
alias disambiguation  
    definition A-1  
    described 3-19  
aliased variables 2-16, 3-12  
aliasing 3-12, A-1  
align heap function 7-66  
allocate and clear memory function 7-46  
allocate memory function 7-66  
allocation  
    definition A-1  
    dynamic 6-5  
    of sections in memory 4-3, 4-10  
alternate directories for include files 2-27  
ANSI A-1  
ANSI C, language 5-1 to 5-30, 7-82, 7-84  
-apd assembler option 2-22  
-api assembler option 2-22  
-ar linker option 4-5  
arc cosine function 7-40  
arc sine function 7-41  
arc tangent function 7-43  
archive library 4-7, A-1  
archiver 1-3, A-1  
--arg\_size linker option 4-5  
- -args linker option 4-5  
argument block 6-15  
array search function 7-45  
array sort function 7-74  
-as assembler option 2-22  
ASCII conversion functions 7-44  
asctime function 7-41, 7-49  
asin function 7-41  
.asm extension 2-8, 2-19  
asm statement  
    C language 5-13, 6-25  
    in optimized code 3-11  
assembler  
    definition A-1  
    description 1-3  
    in the software development flow 1-3  
    options 2-22

- assembly language
  - interfacing with C/C++ language 6-20 to 6-25
  - interlisting with C/C++ language 2-40
  - interrupt routines 6-27
  - modules 6-20 to 6-22
- assembly listing file, creating 2-22
- assembly source debugging 2-18
- assert function 7-42
- assert.h header 7-17, 7-30
- assignment statement A-1
- atan function 7-43
- atan2 function 7-43
- atexit function 7-44, 7-51
- atof function 7-44
- atoi function 7-44
- atol function 7-44
- au assembler option 2-22
- autoincrement addressing 3-26
- autoinitialization
  - at load time
    - definition A-1
    - described 6-40
  - at run time
    - definition A-2
    - described 6-39
  - definition A-1
  - of variables 6-5, 6-36
  - types of 4-9
- automatic inline expansion 3-13
- auxiliary user information file, generating 2-15
- ax assembler option 2-22

## B

- b shell option 2-15
- banner suppressing 2-16
- base-10 logarithm function 7-65
- \_\_big\_endian\_\_ predefined name 2-26
- big-endian
  - changing to little 2-16
  - definition A-2
- BIS, definition A-2
- bit fields 5-3, 5-27, 6-11

- block
  - allocating sections 4-10, 6-2
  - conditionalizing 3-27
  - definition A-2
- boot.asm 6-35
- boot.obj 4-7, 4-9
- branch optimizations 3-19
- break string into token function 7-93
- broken-down time function 7-27, 7-41, 7-61, 7-64, 7-70
- bsearch function 7-45
- .bss section 4-11, 6-3, A-2
- buffer
  - define and associate function 7-78
  - specification function 7-77
- BUFSIZE macro 7-25
- built-in functions 6-34
- byte A-2

## C

- .C extension 2-8, 2-19
- .c extension 2-8, 2-19
- C I/O
  - adding a device 7-14 to 7-16
  - functions and macros summary 7-32
  - implementation 7-5
  - library 7-4 to 7-6
  - low-level routines 7-5
- C language
  - accessing assembler variables 6-23
  - The C Programming Language vi
  - characteristics 5-2
  - compatibility with ANSI C 5-26
  - constants 5-2
  - conversions 5-3
  - data types 5-3, 5-6
  - declarations 5-3
  - expressions 5-3
  - identifiers 5-2
  - interfacing with assembly language 6-20 to 6-25
  - \_\_interrupt keyword 5-8
  - interrupt keyword 5-8
  - interrupt routines 6-26
    - software interrupts 6-28
    - using assembly routines in C 6-27
  - keywords 5-7 to 5-9



- C language (continued)
  - placing assembler statements in 6-25
  - register variables 5-10
  - supported by '470 compiler 5-1 to 5-30
  - volatile keyword 5-9
- c library-build utility option 8-3
- c option
  - linker 4-9
    - and system initialization* 6-35
    - autoinitialization at run time* 6-39
    - description* 4-2, 4-5
    - how it differs from shell option* 4-4
    - specifying run-time-support libraries* 4-7
  - shell
    - description* 2-15
    - disabling the linker with* 4-4
    - how it differs from linker option* 4-4
- C++ language
  - characteristics 5-5
  - embedded C++ mode 5-28
  - exception handling 5-5
  - iostream 5-5
  - run-time type information 5-5
- C++ name demangler
  - description* 1-4, 9-1
  - invoking* 9-2
  - options* 9-2
  - sample usage* 9-3 to 9-4
- C/C++ language
  - accessing assembler constants 6-24
  - interfacing with assembly language 6-20 to 6-25
  - interlisting with assembly 2-40
- C\_DIR environment variable 2-24, 2-27, 8-2
- \_c\_int00 symbol 4-9, 5-17, 6-35
- C\_OPTION environment variable 2-5, 2-15, 2-24, 8-2
- C5X\_C\_OPTION environment variable 2-24 to 2-26
- calendar time function 7-27, 7-49, 7-70, 7-95
- call, macro A-5
- calloc function 6-5, 7-46, 7-57, 7-69
- Cartesian arc tangent function 7-43
- case sensitivity
  - in C language 5-2
  - in filename extensions 2-19
- cassert header 7-17, 7-30
- .cc extension 2-8
- cctype header 7-18, 7-30
- ceil function 7-46
- cerrno header 7-18
- cfloat header 7-19
- change heap size 6-5, 7-46, 7-66, 7-69, 7-75
- char, changing unsigned to signed 2-16
- CHAR\_BIT macro 7-19
- CHAR\_MAX macro 7-19
- CHAR\_MIN macro 7-19
- character
  - conversion functions 7-94
  - read function
    - multiple characters* 7-54
    - single character* 7-53
- character constants 5-27, 6-12
- character sets 5-2
- character-typing functions
  - described* 7-18, 7-62
  - summary* 7-30
- .cinit section 4-10
  - and system initialization* 6-5, 6-35, 6-36
  - and the linking process* 4-9 to 4-10
  - description* 6-2, 6-37 to 6-39
  - format* 6-37
- cinit symbol 4-10
- ciso646 header 7-22
- cl470 -z 4-2
- clear EOF function 7-47
- clearerr function 7-47
- climits header 7-19
- clock function 7-47
- CLOCKS\_PER\_SEC macro 7-47
  - described* 7-27
- close file function 7-52
- close I/O function 7-9
- clp470 command 2-4
- cmath header 7-22
  - summary of functions* 7-31 to 7-33
- Code Composer Studio, and code generation
  - tools* 1-8
- code generator, definition A-2
- \_CODE\_ACCESS macro 7-22
- command file
  - appending to command line* 2-15
  - definition* A-2
- comment, definition A-2

- common logarithm function 7-65
  - common object file format, definition A-2
  - common subexpression elimination 3-21
  - compare strings 7-82, 7-88
  - comparisons to 0, removing 3-29
  - compatibility 5-26 to 5-28
  - compile only 2-16
  - compiler
    - C\_OPTION environment variable 2-24
    - definition A-2
    - description 2-1 to 2-40
    - diagnostic messages 2-30 to 2-34
    - invoking with shell program 2-4
    - limits 5-29 to 5-30
    - options
      - deprecated* 2-23
      - profiling* 2-7
      - summary table* 2-6 to 2-14
      - symbolic debugging* 2-7
    - overview 1-5
    - passing arguments to the linker 4-3
    - sections 4-10
    - summary of options 2-5
  - compiling C/C++ code 2-2
  - concatenate strings function 7-81, 7-87
  - const keyword 5-7
  - .const section 4-10, 6-2, 6-39
    - use to initialize variables 5-25
  - const type qualifier 5-25
  - constants
    - .const section 5-25
    - assembler, accessing from C/C++ 6-24
    - C language 5-2
    - definition A-2
  - constructors, global 4-8, 4-9, 6-35, 6-36
  - control-flow simplification 3-19
  - controlling diagnostic messages 2-32 to 2-33
  - conversions
    - C language 5-3
    - calendar time to Greenwich Mean Time 7-61
    - calendar time to local time 7-49, 7-64
    - case 7-96
    - character 7-18, 7-92, 7-94
    - local time to calendar time 7-70
    - long integer to ASCII 7-65
    - string to number 7-44, 7-92
    - time to string 7-41
    - to ASCII 7-96
  - copy file using `-ahc` assembler option 2-22
  - copy propagation 3-21
  - copy string function 7-83, 7-89
  - cos function 7-48
  - cosh function 7-48
  - cosine function 7-48
  - cost-based register allocation optimization 3-19
  - .cpp extension 2-8, 2-19
  - `-cr` linker option 4-9
    - and system initialization 6-35
    - autoinitialization at load time 6-40
    - description 4-2, 4-5
    - specifying initialization type 4-4, 6-5
    - specifying run-time-support libraries 4-7
  - create temporary file function 7-95
  - create valid filename function 7-96
  - cross-reference lister 1-4
  - cross-reference listing
    - creation 2-22, 2-35
    - definition A-3
  - csetjmp header 7-23, 7-32
  - cstdarg header 7-23, 7-32
  - cstddef header 7-24
  - cstdio header 7-32
    - described 7-25
  - cstdlib header 7-26, 7-35 to 7-36
  - cstring header 7-26, 7-36
  - ctime function 7-49
  - ctime header 7-27 to 7-28, 7-38
  - ctype.h header 7-18, 7-30
  - `__curpc` function 6-34
  - .cxx extension 2-8, 2-19
- ## D
- `-d` shell option 2-15
  - DABT interrupt type 5-17
  - data
    - definition A-3
    - flow optimizations 3-21
    - object representation 6-6 to 6-12
    - types
      - C language* 5-3, 5-6
      - storage* 6-6 to 6-10
  - data abort (DABT) interrupt type 5-17
  - .data section 6-3

- data type storage 6-6 to 6-10
  - data types
    - div\_t 7-26
    - FILE 7-25
    - fpos\_t 7-25
    - jmp\_buf 7-23
    - ldiv\_t 7-26
    - ptrdiff\_t 7-24
    - size\_t 7-24, 7-25
  - \_DATA\_ACCESS macro 7-22
  - DATA\_SECTION pragma 5-14
  - \_\_DATE\_\_ 2-26
  - daylight savings time adjustment 7-27 to 7-28
  - DBL\_ macros 7-20
  - deallocate memory function 7-57
  - debugging
    - optimized code 3-16
    - symbolically A-8
  - declarations, C language 5-3
  - define and associate buffer with stream function 7-78
  - defining variables in assembly language 6-23
  - dem470 command 9-2
  - demangler. *See* C++ name demangler
  - destructors 4-8, 4-9
  - diagnostic identifiers, in raw listing file 2-36
  - diagnostic messages
    - and the assert.h/cassert header file 7-17
    - assert 7-42
    - controlling 2-32
    - description 2-30 to 2-31
    - errors 2-30
    - fatal errors 2-30
    - format 2-30
    - generating 2-32 to 2-33
    - other messages 2-34
    - remarks 2-30
    - suppressing 2-32 to 2-34
    - warnings 2-30
  - difftime function 7-49
  - direct call, definition A-3
  - directives, definition A-3
  - directories, for include files 2-15
  - directory
    - absolute listing files 2-21
    - assembly and cross-reference listing files 2-21
  - directory (continued)
    - assembly files 2-21
    - object files 2-21
    - temporary files 2-21
  - directory specifications 2-21
  - disable, symbolic debugging 2-19
  - div function 7-50
  - div\_t data type 7-26
  - div\_t type 7-26
  - division 5-3
  - documentation, related v to vi
  - dual-state interworking 6-41 to 6-46
  - DUAL\_STATE pragma 5-15
  - duplicate value in memory function 7-68
  - DWARF debug format 2-18
  - dynamic memory allocation
    - definition A-3
    - described 6-5
  - dynamic memory management 7-28
- E**
- e linker option 4-5
  - ea shell option 2-20
  - ec compiler option 2-20
  - EDOM macro 7-18
  - embedded C++ mode 5-28
  - emulator, definition A-3
  - entry points
    - \_c\_int00 4-9
    - definition A-3
    - for C/C++ code 4-9
    - reset vector 4-9
  - enumerator list, trailing comma 5-27
  - environment information function 7-60
  - environment variable
    - C\_DIR 2-24
    - C\_OPTION 2-24
    - C55X\_C\_OPTION 2-24
  - environment variables
    - C\_DIR 2-27, 8-2
    - C\_OPTION 2-5, 2-15, 8-2
    - definition A-3
    - TMP 8-2
  - eo compiler option 2-20
  - eo shell option 2-20
  - EOF macro 7-25

- epilog
    - definition A-3
    - inlining 3-28
  - EPROM programmer 1-3
  - ERANGE macro 7-18
  - errno.h header 7-18
  - error
    - indicators function 7-47
    - mapping function 7-71
  - error messages
    - See also* diagnostic messages
    - assert macro 7-30, 7-42
    - handling with options 2-33, 5-27
    - preprocessor 2-26
  - error reporting 7-18
  - es compiler option 2-20
  - escape sequences 5-2, 5-27
  - exception handling 7-28
  - exception header 7-28
  - executable module, definition A-3
  - exit function 7-44, 7-51
  - exit functions, abort function 7-39
  - exp function 7-51
  - exponential math function 7-22, 7-51
  - expressions
    - C language 5-3
    - definition A-3
    - simplification 3-21
  - extensions
    - changing 2-20
    - nfo 3-5
    - not recognized by shell 2-20
  - external declarations 5-27
  - external symbol, definition A-3
- F**
- f linker option 4-5
  - fa shell option 2-20
  - fabs function 2-38, 7-52
  - \_FAR\_RTS macro 7-22
  - fast interrupt request (FIQ) interrupt type 5-17
  - fatal error 2-30
  - fb compiler option 2-21
  - fc shell option 2-20
  - fclose function 7-52
  - feof function 7-52
  - ferror function 7-53
    - ff compiler option 2-21
  - fflush function 7-53
    - fg compiler option 2-20
  - fgetc function 7-53
  - fgetpos function 7-53
  - fgets function 7-54
  - file
    - copy 2-22
    - extensions, changing 2-20
    - names 2-19
    - options 2-20
    - output 1-6
    - removal function 7-75
    - rename function 7-76
    - type
      - listing, definition* A-5
      - object, definition* A-6
      - source, definition* A-7
  - FILE data type 7-25
  - file.h header 7-2, 7-18
  - file-position indicator functions
    - get current file position indicator 7-59
    - position at beginning of file 7-76
    - set file position indicator 7-58, 7-59
    - store object at file-position indicator 7-53
  - \_\_FILE\_\_ 2-26
  - file-level optimization 3-4, A-4
  - filename, generate function 7-96
  - FILENAME\_MAX macro 7-25
  - find first occurrence of
    - byte 7-67
    - character 7-82
  - find last occurrence of character 7-90
  - find matching character 7-90, 7-91
  - find matching string 7-91
  - find string length 7-86
  - find unmatching characters 7-84
  - FIQ interrupt type 5-17
  - float.h header 7-19
  - floating-point math functions
    - acos 7-40
    - asin 7-41
    - atan 7-43
    - atan2 7-43
    - ceil 7-46

## floating-point math functions (continued)

- cos 7-48
- cosh 7-48
- defined in the math.h/cmath header file 7-22
- exp 7-51
- fabs 7-52
- floor 7-54
- fmod 7-55
- ldexp 7-63
- log 7-64
- log10 7-65
- modf 7-71
- pow 7-72
- sin 7-79
- sinh 7-79
- sqrt 7-80
- tan 7-94
- tanh 7-94

floating-point remainder 7-55

floating-point, summary of functions 7-31 to 7-33

floor function 7-54

FLT\_ macros 7-20

flush I/O buffer function 7-53

fmod function 7-55

–fo shell option 2-20

fopen function 7-55

FOPEN\_MAX macro 7-25

format time function 7-85

format.h header 7-2

FP register 6-13

–fp shell option 2-20

fpos\_t data type 7-25

fprintf function 7-55

fputc function 7-56

fputs function 7-56

–fr compiler option 2-21

fraction and exponent function 7-58

fread function 7-56

free function 7-57

freopen function 7-57

frexp function 7-58

–fs compiler option 2-21

fscanf function 7-58

fseek function 7-58

fsetpos function 7-59

–ft compiler option 2-21

ftell function 7-59

FUNC\_EXT\_CALLED pragma 5-16

- use with –pm option 3-8

function calls

- conventions 6-15 to 6-19
- using the stack 6-4

function inlining 2-38 to 2-39, A-4

function prototypes 5-26

fwrite function 7-59

**G**

–g compiler option 2-18

–g option

- linker 4-5
- shell 2-18

general-purpose registers

- 32-bit data 6-7, 6-8
- double-precision floating-point data 6-9

general utility functions

- abs 7-40
- atexit 7-44
- atof 7-44
- atoi 7-44
- atol 7-44
- bsearch 7-45
- calloc 7-46
- described 7-26
- div 7-50
- exit 7-51
- free 7-57
- labs 7-40
- ldiv 7-50
- ltoa 7-65
- malloc 7-66
- minit 7-69
- qsort 7-74
- rand 7-74
- realloc 7-75
- srand 7-74
- strtod 7-92
- strtol 7-92
- strtoul 7-92
- summary 7-35 to 7-36

generating

- 16-bit code 2-16
- linknames 5-23
- symbolic debugging directives 2-18, 2-19
- valid filename function 7-96

- get file-position function 7-59
- getc function 7-60
- getchar function 7-60
- getenv function 7-60
- gets function 7-61
- global
  - definition A-4
  - object, constructors 4-9
  - variables
    - initializing* 4-9, 5-24
    - reserved space* 6-2
- global constructors 4-8, 4-9, 6-35, 6-36
- global destructors 4-8, 4-9
- gmtime function 7-61
- Greenwich Mean Time 7-61
- Gregorian time 7-27

## H

- -h library-build utility option 8-3
- h option
  - demangler 9-2
  - linker 4-5
- header files 7-16 to 7-28
  - ciso646 7-22
  - cstdio header 7-25
  - file.h header 7-18
  - iso646.h 7-22
  - linkage.h header 7-22
  - stdint.h 7-24
  - stdio.h header 7-25
- heap
  - align function 7-66
  - described 6-5
  - reserved space 6-3
- heap linker option 4-5, 7-4, 7-46, 7-66, 7-69, 7-75
- hex-conversion utility 1-3
- HUGE\_VAL 7-22
- hyperbolic math functions
  - cosine 7-48
  - defined by math.h/cmath header 7-22
  - sine 7-79
  - tangent 7-94

## I

- i option
  - linker 4-5
  - shell 2-15, 2-27
- I/O
  - functions
    - close* 7-9
    - flush buffer* 7-53
    - lseek* 7-10
    - open* 7-11
    - read* 7-12
    - rename* 7-12
    - unlink* 7-13
    - write* 7-13
  - low-level definitions 7-18
- \_IDECL macro 7-22
- identifiers, C language 5-2
- implementation-defined behavior 5-2 to 5-4
- #include files 2-15, 2-26, 2-27
- indirect call, definition A-4
- induction variables 3-24
- initialization
  - of variables
    - at load time* 6-5, 6-40
    - at run time* 6-5, 6-39
    - global* 4-9, 5-24, 6-5
    - static* 5-24, 6-5
  - tables 6-37 to 6-39
    - See also .cinit section; .pinit section*
  - types for global variables 4-9
- initialized sections 4-10
  - .cinit 6-2
  - .const 6-2
  - definition A-4
  - description 6-2
  - .pinit 6-2
  - .text 6-2
- \_INLINE 2-26
- inline
  - assembly language 6-25
  - automatic expansion 2-38, 3-13
  - declaring functions as 2-39
  - definition-controlled 2-39
  - disabling 2-39
  - epilog 3-28
  - function, definition A-4
  - function expansion 2-38 to 2-39
  - intrinsic operators 2-38

- inline (continued)
    - keyword and alternate keyword `__inline` 2-39
    - run-time-support functions 3-23
    - strict ANSI C compatibility 2-39
  - `__inline` keyword 2-39
  - input/output definitions 7-18
  - `int_fastN_t` integer type 7-24
  - `int_leastN_t` integer type 7-24
  - `INT_MAX` macro 7-19
  - `INT_ML` macro 7-19
  - integer division function 7-50
  - integer type range limits 7-19
  - integrated preprocessor, definition A-4
  - interfacing C/C++ and assembly language 6-20 to 6-25
  - interlist feature, using with the optimizer 3-14
  - interlist utility
    - definition A-4
    - invoking 2-17, 2-40
    - used with the optimizer 3-14
  - interrupt
    - handling 6-26 to 6-29
      - assembly language interrupt routines* 6-27
      - saving registers* 6-26
      - software interrupts* 6-28
    - routines
      - C/C++ language* 6-26
      - mapping* 6-27
      - prefixing* 6-27
    - types 5-17
    - vectors 6-27
  - `__interrupt` keyword 5-8
  - `INTERRUPT` pragma 5-17
  - interrupt request (IRQ) interrupt type 5-17
  - `intmax_t` integer type 7-24
  - `INTN_C` macro 7-24
  - `intN_t` integer type 7-24
  - `intptr_t` integer type 7-24
  - intrinsic operators 2-38
  - intrinsic run-time-support routines 6-30 to 6-33
  - inverse tangent of  $y/x$  function 7-43
  - invoking the
    - compiler, with the shell 2-2
    - interlist utility, with the shell 2-40
    - library-build utility 8-2
  - linker
    - separately* 4-2
    - with compiler* 4-2
    - with the shell* 2-2, 4-3
    - optimizer, with shell options 3-2
    - shell program 2-4
  - iostream support 5-5
  - IRQ interrupt type 5-17
  - `isalnum` function 7-62
  - `isalpha` function 7-62
  - `isascii` function 7-62
  - `isctrl` function 7-62
  - `isdigit` function 7-62
  - `isgraph` function 7-62
  - `islower` function 7-62
  - ISO C
    - enabling embedded C++ mode 5-28
    - standard overview 1-5
  - `iso646.h` header 7-22
  - `isprint` function 7-62
  - `ispunct` function 7-62
  - `isspace` function 7-62
  - `isupper` function 7-62
  - `isxdigit` function 7-62
  - `isxxx` function 7-18, 7-62
- ## J
- `-j` linker option 4-5
  - `jmp_buf` data type 7-23
  - `jmpbuf` type 7-23
- ## K
- `-k` library-build utility option 8-3
  - `-k` shell option 2-15
  - K&R C vi
    - compatibility 5-2 to 5-4, 5-26
    - definition A-4
    - mode 5-26
  - keywords
    - C language keywords 5-7 to 5-9
    - `const` 5-7
    - `inline` 2-39
    - `__inline` 2-39
    - `__interrupt` 5-8
    - `interrupt` 5-8
    - `volatile` 5-9

**L**

- l option
  - library-build utility 8-2
  - linker 4-2, 4-6, 4-7
- L\_tmpnam macro 7-25
- labels
  - definition A-5
  - retaining 2-22
- labs function 2-38, 7-40
- LDBL\_ macros 7-20
- ldexp function 7-63
- ldiv function 7-50
- ldiv\_t data type 7-26
- ldiv\_t type 7-26
- library
  - building 7-3
  - C I/O 7-4 to 7-6
  - modifying a function 7-3
  - object A-6
  - run-time support A-7
  - run-time-support 7-2 to 7-3
- library-build utility
  - description 1-3
  - invoking 8-2
  - optional object library 8-2
  - options 8-2, 8-3 to 8-5
- limits
  - compiler 5-29 to 5-30
  - floating-point types 7-20
  - integer types 7-19
- limits.h header 7-19
- \_\_LINE\_\_ 2-26
- linkage.h header 7-22
- linker
  - arguments passed from the compiler 4-3
  - command file 4-11 to 4-13
  - definition A-5
  - description 1-3
  - disabling 4-4
  - in the software development flow 1-3
  - invoking, with the shell 2-4, 4-3
  - options summary 4-5 to 4-6
  - suppressing 2-15, 4-4
- linking
  - C/C++ code 4-1 to 4-14
  - controlling 4-7
  - requirements for 4-7 to 4-13
  - with run-time-support libraries 4-7
  - with the compiler 4-2
  - with the shell program 4-3
- linknames
  - and C++ name mangling 5-23
  - and interrupts 6-29
  - generating 5-23
- listing file
  - creating cross-reference 2-22
  - definition A-5
- \_\_little\_endian\_\_ predefined name 2-26
- little-endian
  - definition A-5
  - producing 2-16
- load-time autoinitialization 6-40
- loader 4-9 to 4-10, 5-24, A-5
- local register variables 5-10
- local time function 7-27 to 7-28, 7-49
- localtime function 7-49, 7-64, 7-70
- log function 7-64
- log10 function 7-65
- LONG\_MAX macro 7-19
- LONG\_MIN macro 7-19
- longjmp function 7-23, 7-77
- loop rotation 3-24
- loop unrolling, definition A-5
- loop-invariant optimization 3-24
- low-level I/O functions 7-18
- lseek I/O function 7-10
- ltoa function 7-65

**M**

- m linker option 4-6
- ma shell option 2-16
- macro
  - call A-5
  - definition A-5
  - definitions 2-26 to 2-27
  - expansions 2-26 to 2-27, A-5
  - that supplies floating-point range limits 7-20
  - that supplies integer type range limits 7-19



## macros

- `_CODE_ACCESS` 7-22
- `_DATA_ACCESS` 7-22
- `_FAR_RTS` 7-22
- `_IDDECL` 7-22
- `BUFSIZ` 7-25
- `CLOCKS_PER_SEC` 7-27
- `EOF` 7-25
- `FILENAME_MAX` 7-25
- `FOPEN_MAX` 7-25
- `L_tmpnam` 7-25
- `NULL` 7-24, 7-25
- `offsetof` 7-24
- `SEEK_CUR` 7-25
- `SEEK_END` 7-25
- `SEEK_SET` 7-25
- `setjmp` 7-23
- `stden` 7-25
- `stdin` 7-25
- `stdout` 7-25
- `TMP_MAX` 7-25

`malloc` function 6-5, 7-57, 7-66, 7-69

`map`, memory 6-2, A-5

`map` error number 7-71

`map` file A-5

`mapping`, interrupt routines 6-27

`math.h` header 7-22, 7-31

summary of functions 7-31 to 7-33

`-mc` shell option 2-16

`-md` shell option 6-43

`-me` shell option 2-16

`memalign` function 7-66

`memchr` function 7-67

`memcmp` function 7-67

`memcpy` function 7-68

`memmove` function 7-68

`memory` block copy 7-68

`memory` compare 7-67

`MEMORY` directive 4-3

`memory` management functions

`calloc` 7-46

`free` 7-57

`malloc` 7-66

`minit` 7-69

`realloc` 7-75

`memory` map 6-2, A-5

## memory model

dynamic memory allocation 6-5, 7-28

sections 6-2 to 6-3

stack 6-4

variable initialization 6-5

`memory` pool 6-5, 7-66

`memset` function 7-68

`-mf` shell option 2-16, 7-3

`minit` function 7-69

`mkp470` 8-2

`mktime` function 7-70

`-mn` shell option 2-16

`modf` function 7-71

`module`, output A-6

`modulus` 5-3

`-mt` shell option 2-16, 6-44

`multibyte` characters 5-2

`multiply` by power of 2 function 7-63

`MUST_ITERATE` pragma 5-18

## N

`-n` option, shell 2-16

`name` demangler

See also C++ name demangler

description 1-4

`natural` logarithm function 7-64

`NDEBUG` macro 7-17, 7-42

`new` header 7-28

`new_handler` type 7-28

`.nfo` extension 3-5

`nonlocal` jumps 7-23, 7-32, 7-77

`nonstandard` header files in run-time-support

libraries 7-2

`NULL` macro 7-24, 7-25

## O

`-o` option

demangler 9-2

linker 4-6

shell 3-2, 7-3

`-o3` with `-ol` 3-4

`-o3` with `-on` 3-5

`-o3` with `-pi` 2-39

`.o*` extension 2-8

- .obj extension 2-19
  - changing 2-20
- object file, definition A-6
- object library A-6
- object representation in run-time environment 6-6 to 6-12
- offsetof macro 7-24
- oi shell option 3-13
- ol shell option 3-4
- on shell option 3-5
- online shell option summary 2-5
- op shell option 3-7 to 3-9
- open file function 7-55, 7-57
- open I/O function 7-11
- operand, definition A-6
- optimization
  - accessing aliased variables 3-12
  - alias disambiguation 3-19
  - autoincrement addressing 3-26
  - block conditionalizing 3-27
  - branch 3-19
  - control-flow simplification 3-19
  - cost-based register allocation 3-19
  - data flow 3-21
  - epilog inlining 3-28
  - expression simplification 3-21
  - file-level, definition 3-4, A-4
  - induction variables 3-24
  - information file options 3-5
  - inline expansion 3-23
  - list of 3-18 to 3-30
  - loop rotation 3-24
  - loop-invariant code motion 3-24
  - program-level
    - definition A-6
    - FUNC\_EXT\_CALLED* pragma 5-16
  - removing comparisons to 0 3-29
  - speed over size 2-16
  - strength reduction 3-24
  - tail merging 3-24
- optimizations
  - controlling the level of 3-7
  - levels 3-2
  - program-level, described 3-6
- optimized code, debugging 3-16
- optimizer
  - definition A-6
  - invoking, with shell options 3-2
  - special considerations 3-11
  - summary of options 2-12
  - use with debugger 2-16
- options
  - assembler 2-13, 2-22
  - C++ name demangler utility 9-2
  - changing file extensions 2-7, 2-20 to 2-27
  - compiler 2-5 to 2-14
  - conventions 2-5
  - definition A-6
  - diagnostics 2-10, 2-32
  - frequently used, compiler 2-15 to 2-18
  - library-build utility 8-2, 8-3
  - linker 2-13, 4-5 to 4-6
  - optimization 2-12, 3-2 to 3-9
  - parsing 2-9
  - preprocessor 2-10, 2-28 to 2-29
  - run-time model 2-11, 2-16, 6-43, 6-44
  - shell 2-6 to 2-14
  - specifying directories 2-8, 2-21
  - specifying files 2-8, 2-20
  - summary table 2-6 to 2-14
- output, overview of files 1-6
- output module A-6
- output section A-6
- overflow, run-6-35

## P

- PABT interrupt type 5-17
- parser, definition A-6
- pdel shell option 2-32
- pden shell option 2-32
- pdf shell option 2-32
- pdr shell option 2-32
- pds shell option 2-32
- pdse shell option 2-32
- pdsr shell option 2-32
- pdsr compiler option 2-32
- pdv shell option 2-33
- pdw shell option 2-33
- pe compiler option 5-28
- perror function 7-71

- pi shell option 2-39
- .pinit section 4-10
  - and global constructors 4-9, 6-35, 6-36, 6-38 to 6-39
  - and system initialization 6-35
  - and the linking process 4-10
  - description 6-2
  - format 6-38
- pinit symbol 4-10
- pk shell option 5-26 to 5-27
- pm shell option 3-6
- pointer combinations 5-27
- polar arc tangent function 7-43
- position file indicator function 7-76
- pow function 7-72
- ppa shell option 2-28
- ppc shell option 2-28
- ppd shell option 2-29
- ppi shell option 2-29
- ppl shell option 2-29
- ppo shell option 2-28
- pr shell option 5-28
- #pragma directive 5-4
  - DATA\_SECTION 5-14
  - definition A-6
  - description 5-14 to 5-22
  - DUAL\_STATE 5-15
  - FUNC\_EXT\_CALLED 5-16
  - INTERRUPT 5-17
  - SWI\_ALIAS 5-20
  - syntax conventions 5-14
  - TASK 5-20
- pragma directives
  - MUST\_ITERATE 5-18
  - UNROLL 5-22
- predefined names 2-26
  - \_\_TIME\_\_ 2-26
  - \_\_DATE\_\_ 2-26
  - \_\_FILE\_\_ 2-26
  - \_\_LINE\_\_ 2-26
  - ad assembler option 2-22
  - \_INLINE 2-26
  - \_\_TI\_COMPILER\_VERSION\_\_ 2-26
  - undefining with –au assembler option 2-22
- prefetch abort (PABT) interrupt type 5-17
- preinitialized 5-24
- preprocessed listing file
  - assembly dependency lines 2-22
  - assembly include files 2-22
- preprocessor
  - controlling 2-26 to 2-29
  - definition A-6
  - directives 2-26, 5-4, 5-27
  - error messages 2-26
  - options 2-28 to 2-29
  - predefining name 2-15
  - stand-alone A-7
  - symbols 2-26
- printf function 7-72
- processor time function 7-47
- profile:breakpt compiler option 2-18
- profile:power compiler option 2-18
- profiling optimized code 3-17
- program termination functions
  - abort function 7-39
  - atexit 7-44
  - exit 7-51
- program-level optimization
  - controlling 3-7
  - definition A-6
  - performing 3-6
- progress information suppressing 2-16
- ps shell option 5-28
- pseudorandom 7-74
- ptrdiff\_t data type 5-3, 7-24
- putc function 7-72
- putchar function 7-73
- puts function 7-73

## Q

- –q library-build utility option 8-3
- q option
  - linker 4-6
  - shell 2-16
- qsort function 7-74

## R

- r linker option 4-6
- R13 register (stack pointer) 6-4
- rand function 7-74
- RAND\_MAX macro 7-26

- random integer function 7-74
- range limits in
  - floating-point types 7-20
  - integer types 7-19
- raw listing file
  - generating with `-pl` option 2-36
  - identifiers 2-36
- read
  - character functions
    - multiple characters* 7-54
    - next character* 7-53, 7-60
    - single character* 7-53
  - stream functions
    - from standard input* 7-76
    - from string to array* 7-56
    - string* 7-58, 7-81
- read function 7-61
- read I/O function 7-12
- realloc function 6-5, 7-57, 7-69, 7-75
- redundant assignment elimination 3-21
- register conventions 6-13 to 6-14
- register save area 6-15
- register storage class 5-3
- registers
  - save-on-call 6-15
  - save-on-entry 6-15
  - saving during interrupts 6-26
  - saving upon entry 6-26
- related documentation v to vi
- relaxed ANSI mode 5-28
- relocation, definition A-7
- remarks 2-30
- remove file function 7-75
- removing comparisons to 0 3-29
- rename function 7-76
- rename I/O function 7-12
- reset dynamic memory pool 7-69
- RESET interrupt type 5-17
- return from main 4-8
- rewind function 7-76
- rtsc.src 1-3, 7-2, 7-26
- rtsc\_16.lib 1-3, 4-2
- rtsc\_32.lib 1-3, 4-2
- rtscpp.src 1-3, 7-2, 7-26
- rtscpp\_16.lib 1-3, 4-2

- rtscpp\_32.lib 1-3, 4-2
- rtti.h header 7-2
- run-time environment
  - defining variables in assembly language 6-23
  - definition A-7
  - function call conventions 6-15 to 6-19
  - inline assembly language 6-25
  - interfacing C/C++ with assembly language 6-20 to 6-25
  - interrupt handling 6-26 to 6-29
    - assembly language interrupt routines* 6-27
    - saving registers* 6-26
    - software interrupts* 6-28
  - memory model
    - during autoinitialization* 6-5
    - dynamic memory allocation* 6-5
    - sections* 6-2
  - register conventions 6-13 to 6-14
  - stack 6-4
  - system initialization 6-35 to 6-40
- run-time initialization of variables 6-5, 6-39
- run-time type information 5-5
- run-time-support
  - functions
    - definition* A-7
    - descriptions* 7-39 to 7-99
    - introduction* 7-1
    - summary* 7-29 to 7-38
  - intrinsic arithmetic and conversion routines 6-30 to 6-33
  - libraries
    - definition* A-7
    - described* 1-3, 7-2, 8-1
    - linking with* 4-2, 4-7
  - library function inline expansion 3-23
  - macros, summary 7-29 to 7-38
- `__run_address_check` function 6-34

## S

- `.s` extension 2-19
- `-s` option
  - compiler 2-40
  - linker 4-6
  - shell 2-17, 2-40
- `.s*` extension 2-8
- save-on-call registers 6-15
- save-on-entry registers 6-15

- saving registers
  - during interrupts 6-26
  - upon entry 6-26
- scanf function 7-76
- SCHAR\_MAX macro 7-19
- SCHAR\_MIN macro 7-19
- sdp470 command 10-2
- search path modification 2-27
- searches 7-45
- section, .const, initializing 5-25
- sections
  - allocating 4-10, 6-2
  - .bss 6-3
  - .cinit 6-2, 6-3, 6-35, 6-36, 6-37 to 6-39
  - .const 6-2, 6-3
  - .data 6-3
  - definition A-7
  - described 4-10, 6-2 to 6-3
  - header A-7
  - initialized 6-2, A-4
  - output A-6
  - .pinit 4-9, 6-2, 6-3, 6-35, 6-36, 6-38 to 6-39
  - .stack 6-3
  - .system 6-3
  - .text 6-2, 6-3
  - uninitialized 6-3, A-8
- SECTIONS directive 4-3
- SEEK\_CUR macro 7-25
- SEEK\_END macro 7-25
- SEEK\_SET macro 7-25
- set file-position functions
  - fseek function 7-58
  - fsetpos function 7-59
- set\_new\_handler() function 7-28
- setbuf function 7-77
- setjmp function 7-77
- setjmp macro 7-23
- setjmp.h header 7-23, 7-32
- setvbuf function 7-78
- shell program
  - assembler options 2-22
  - compile only 2-16
  - definition A-7
  - diagnostic options 2-32 to 2-33
  - directory specifier options 2-21
  - enabling linking 2-18
  - file specifier options 2-20
  - shell program (continued)
    - general options 2-15 to 2-18
    - generate auxiliary user information file 2-15
    - invoking the 2-4
    - keeping the assembly language file 2-15
    - overview 2-2 to 2-3
    - preprocessor options 2-28 to 2-29
    - summary of options
      - listing online* 2-5
      - table* 2-6 to 2-14
  - shift 5-3
  - SHRT\_MAX macro 7-19
  - SHRT\_MIN macro 7-19
  - signed integer and fraction function 7-71
  - \_\_signed\_chars\_\_ predefined name 2-26
  - sin function 7-79
  - sine 7-79
  - sinh function 7-79
  - \_\_16bis\_\_ predefined name 2-26
  - size\_t data type 5-3, 7-24, 7-25
  - software development tools 1-2 to 1-4
  - software interrupt 6-28
  - software interrupt (SWI) interrupt type 5-17
  - software stack. *See* stack
  - sorts function 7-74
  - source file
    - definition A-7
    - extensions 2-20
  - specify buffer for stream function 7-77
  - specifying directories 2-21
  - sprintf function 7-80
  - sqrt function 7-80
  - square root function 7-80
  - srand function 7-74
  - ss compiler option 3-14
  - ss shell option 2-17, 3-14
  - sscanf function 7-81
  - STABS debugging format 2-19
  - stack
    - changing the size 6-4, 6-35
    - management 6-4
    - overflow, run-time stack 6-4, 6-35
    - pointer 6-35
    - reserved space 6-3
    - stack linker option 4-6, 6-4, 6-35
    - .stack section 4-11, 6-3

- \_\_STACK\_SIZE constant 6-4
- stand-alone preprocessor A-7
- static
  - definition A-7
  - reserved space 6-3
  - variables 5-24
- static depth profiler, description 1-4
- stdarg.h header 7-23, 7-32
- stddef.h header 7-24
- stden macro 7-25
- stdexcept header 7-28
- stdin macro 7-25
- stdint.h header 7-24
- stdio.h header 7-32
  - described 7-25
- stdlib.h header 7-26, 7-35 to 7-36
- stdout macro 7-25
- storage class A-7
- store object function 7-53
- strcat function 7-81
- strchr function 7-82
- strcmp function 7-82
- strcoll function 7-82
- strcpy function 7-83
- strcspn function 7-84
- strength reduction optimization 3-24
- strerror function 7-84
- strftime function 7-85
- strict ANSI mode 5-28
- string constants 6-12
- string copy 7-89
- string functions 7-26, 7-36
- string.h header 7-26, 7-36
- strlen function 7-86
- strncat function 7-87
- strncmp function 7-88
- strncpy function 7-89
- strpbrk function 7-90
- strrchr function 7-90
- strspn function 7-91
- strstr function 7-91
- strtod function 7-92
- strtok function 7-93
- strtol function 7-92
- strtoul function 7-92
- struct\_tm 7-27
- structure, definition A-8
- structure members 5-3
- strxfrm function 7-94
- STYP\_CPY flag 4-10
- suppressing
  - banners 2-16
  - diagnostic messages 2-32 to 2-34
  - progress information 2-16
- SWI interrupt type 5-17
- SWI\_ALIAS pragma 5-20
- symbol A-8
- symbol table A-8
- symbolic, debugging, generating directives 2-18
- symbolic debugging
  - cross-reference, creating 2-22
  - definition A-8
  - DWARF directives 2-18
  - disabling 2-19
  - minimal (default) 2-19
  - using STABS format 2-19
- symbols
  - assembler-defined 2-22
  - undefining assembler-defined symbols 2-22
- symdebug:coff compiler option 2-19
- symdebug:dwarf compiler option 2-18
- symdebug:none compiler option 2-19
- symdebug:skeletal compiler option 2-19
- .system section 6-3
- system section 4-11
- \_\_SYSTEM\_SIZE 6-5
  - memory management 7-26
- system constraints
  - \_\_STACK\_SIZE 6-4
  - \_\_SYSTEM\_SIZE 6-5
- system initialization 6-35 to 6-40
  - autoinitialization 6-36
  - stack 6-35
- system reset (RESET) interrupt type 5-17

## T

- tail merging 3-24
- tan function 7-94
- tangent function 7-94
- tanh function 7-94

target system A-8  
 TASK pragma 5-20  
 temporary file creation function 7-95  
 tentative definition 5-27  
 test EOF indicator 7-52  
 test error function 7-53  
 text, definition A-8  
 .text section 4-10, 6-2  
 \_\_32bis\_\_ predefined name 2-26  
 \_\_TI\_COMPILER\_VERSION\_\_ 2-26  
 time functions  
     asctime 7-41  
     clock 7-47  
     ctime 7-49  
     difftime 7-49  
     gmtime 7-61  
     localtime 7-64  
     mktime 7-70  
     strftime 7-85  
     summary table 7-38  
     time 7-95  
 time.h header 7-27 to 7-28, 7-38  
 \_\_TIME\_\_ 2-26  
 time\_t data type 7-27 to 7-28  
 tm structure 7-27  
 TMP environment variable 8-2  
 TMP\_MAX macro 7-25  
 tmpfile function 7-95  
 tmpnam function 7-96  
 \_\_TMS470\_\_ predefined name 2-26  
 toascii function 7-96  
 tokens 7-93  
 tolower function 7-96  
 toupper function 7-96  
 trailing comma, enumerator list 5-27  
 trailing tokens, preprocessor directives 5-27  
 trigonometric math function 7-22  
 trigraph, sequence, definition A-8  
 type\_info structure 7-28  
 typeinfo header 7-28

## U

-u option  
     demangler 9-2  
     linker 4-6  
     shell 2-17  
 UCHAR\_MAX macro 7-19  
 UDEF interrupt type 5-17  
 uint\_fastN\_t unsigned integer type 7-24  
 uint\_leastN\_t unsigned integer type 7-24  
 UINT\_MAX macro 7-19  
 uintmax\_t unsigned integer type 7-24  
 UINTN\_C macro 7-24  
 uintN\_t unsigned integer type 7-24  
 uintprt\_t unsigned integer type 7-24  
 ULONG\_MAX macro 7-19  
 undefine constant 2-17  
 undefined instruction (UDEF) interrupt type 5-17  
 ungetc function 7-97  
 uninitialized sections 4-10  
     .bss 6-3  
     definition A-8  
     description 6-3  
     .stack 6-3  
     .systemem 6-3  
 unlink I/O function 7-13  
 UNROLL pragma 5-22  
 unsigned, definition A-8  
 unsigned char, changing to signed 2-16  
 \_\_unsigned\_chars\_\_ predefined name 2-26  
 USHRT\_MAX macro 7-19  
 utilities, overview 1-7

## V

-v demangler option 9-2  
 - -v library-build utility option 8-3  
 va\_arg function 7-97  
 va\_end function 7-97  
 va\_start function 7-97  
 values.h header 7-2  
 variable, definition A-8  
 variable argument functions and macros 7-23, 7-32, 7-97  
 variables, assembler, accessing from C 6-23  
 veneer, definition A-8  
 vfprintf function 7-98  
 vprintf function 7-98

vsprintf function 7-99

vtbl.h header 7-2

## W

-w linker option 4-6

warning messages 2-30, 5-27

wildcard 2-19

write block of data function 7-59

write character function 7-56, 7-72, 7-73

write functions

fprintf 7-55

fputc 7-56

fputs 7-56

fwrite 7-59

printf 7-72

putc 7-72

putchar 7-73

puts 7-73

sprintf 7-80

write functions (continued)

ungetc 7-97

vfprintf 7-98

vprintf 7-98

vsprintf 7-99

write I/O function 7-13

write stream function 7-55, 7-80, 7-99

write string function 7-56

write to standard output function 7-72, 7-73, 7-98

write to stream function 7-97, 7-98

## X

-x linker option 4-6, 7-2

## Z

-z compiler option 4-2 to 4-4

-z shell option 2-2, 2-4, 2-18, 4-3 to 4-4

0, removing comparisons to 3-29