TMS320 Floating-Point DSP Optimizing C Compiler User's Guide

2576391-9761 revision B
February 1995

TEXAS INSTRUMENTS

Preface

Read This First

About This Manual

The *TMS320 Floating-Point DSP Optimizing C Compiler User's Guide* tells you how to use these compiler tools:

- ❑ Compiler
- ❑ Source interlist utility
- ❑ Optimizer
- ❑ Preprocessor
- ❑ Library-build utility

This compiler accepts American National Standards Institute (ANSI) standard C source code and produces assembly language source code for the TMS320C3x/4x devices.

This user's guide discusses the characteristics of the TMS320C3x/4x optimizing C compiler. It assumes that you already know how to write C programs. *The C Programming Language* (second edition), by Brian W. Kernighan and Dennis M. Ritchie, describes C based on the ANSI C standard. Use the Kernighan and Ritchie book as a supplement to this manual.

Before you can use this book, you should read the *TMS320 Floating-Point DSP Code Generation Tools Getting Started* to install the C compiler tools.

How to Use This Manual

The goal of this book is to help you learn how to use the Texas Instruments C compiler tools specifically designed for the TMS320C3x/4x devices. This book is structured in distinct parts:

- **Introductory information,** in chapter one, provides an overview of the TMS320C3x/4x development tools.
- **Compiler description**, in chapter two, describes how to operate the C compiler and the shell program, and discusses specific characteristics of the C compiler as they relate to the ANSI C specification. It contains technical information on the TMS320C3x/4x architecture and includes information needed for interfacing assembly language to C programs. It describes libraries and header files in addition to the macros, functions, and types they declare. Finally, it describes the library-build utility.
- **Reference material**, in chapters three through six and the glossary, provides supplementary information on TMS320C3x/4x specific optimizations, and definitions of terms used in the book.

Notational Conventions

This document uses the following conventions.

- Program listings, program examples, and interactive displays are shown in a `special typeface` similar to a typewriter's. 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 program listing:

```
0011  0005  0001         .field    1, 2
0012  0005  0003         .field    3, 4
0013  0005  0006         .field    6, 3
0014  0006               .even
```

- In syntax descriptions, the instruction, command, or directive is in a **bold face font** and parameters are in *italics.* Portions of a syntax that are in **bold face** should be entered as shown; portions of a syntax that are in *italics* describe the type of information that should be entered. Syntax that will be entered on a command line is centered in a bounded box. Syntax that will be used in a text file is left justified in an unbounded box. Here is an example of a directive syntax:

  #include "*filename*"

The **#include** preprocessor directive has one required parameter, *filename*. The filename must be enclosed in double quotes or angle brackets.

- Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you don't enter the brackets themselves. Here's an example of a command that has an optional parameter:

```
clist asmfile [outfile] [–options]
```

The **clist** command has three parameters. The first parameter, *asmfile*, is required. The second and third parameters, *outfile* and *–options*, are optional. If you omit the outfile, the file has the same name as the assembly file with the extension .cl. Options are preceded by a hyphen.

Square brackets are also used as part of the pathname specification for VMS pathnames; in this case, the brackets are actually part of the pathname (they are not optional).

Related Documentation

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

You may find these documents helpful as well:

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

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

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

American National Standards Institute C Specification, American National Standard for Information Systems—Programming Language C x3.159–1989 (ANSI standard for C)

Related Documentation from Texas Instruments

The following books, which describe the TMS320C3x/C4x and related support tools, are available from Texas Instruments. To obtain TI literature, please call the Texas Instruments Literature Response Center (LRC) at 1 (800)–477–8924, and identify the book by its title and literature number.

TMS320C3x User's Guide (literature number SPRU031) describes the 'C3x 32-bit floating-point microprocessor (developed for digital signal processing as well as general applications), its architecture, internal register structure, instruction set, pipeline, specifications, and DMA and serial port operation. Software and hardware applications are included.

TMS320C4x User's Guide (literature number SPRU063) describes the 'C4x 32-bit floating-point processor, developed for digital signal processing as well as parallel processing applications. Covered are its architecture, internal register structure, instruction set, pipeline, specifications, and operation of its six DMA channels and six communication ports. Software and hardware applications are included.

TMS320C3x C Source Debugger User's Guide (literature number SPRU053) tells you how to invoke the 'C3x emulator, evaluation module, and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints, and includes a tutorial that introduces basic debugger functionality.

TMS320C30 Evaluation Module Technical Reference (literature number SPRU069) describes board-level operation of the TMS320C30 EVM.

TMS320C4x C Source Debugger Installation Guide (literature number SPRU079) tells you how to install the C source debugger interface along with the 'C4x emulator (using the OS/2 and DOS operating systems) and simulator (using the DOS, VMS, and SunOS operating systems). It also covers specifications for connecting your target system to the emulator.

TMS320 Floating-Point DSP Assembly Language Tools User's Guide (literature number SPRU035) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the 'C3x and 'C4x generations of devices.

If You Need Assistance. . .

If you want to. . .	**Do this. . .**
Request more information about Texas Instruments digital signal processing (DSP) products	Call the DSP hotline: **(713) 274–2320** Write to: Texas Instruments Incorporated Market Communications Mgr, MS 736 P.O. Box 1443 Houston, Texas 77251–1443
Order Texas Instruments documentation	Call TI Literature Response Center: **(800) 477–8924**
Ask questions about product operation or report suspected problems	Call the DSP hotline: **(713) 274–2320** or fax your request: **(713) 274–2324**
Report mistakes or make comments about this, or any other TI documentation Please mention the full title of the book and the date of publication (from the spine and/or front cover) in your correspondence.	Send your comments to **comments@books.sc.ti.com** Texas Instruments Incorporated Technical Publications Mgr, MS 702 P.O. Box 1443 Houston, Texas 77251–1443

Trademarks

MS-DOS is a registered trademark of Microsoft Corp.

PC-DOS is a trademark of International Business Machines Corp.

SPARC is a trademark of SPARC International, Inc.

Sun-OS and SunWorkstation are trademarks of Sun Microsystems, Inc.

XDS is a trademark of Texas Instruments Incorporated.

Contents

Figures

Tables

Examples

Notes

Chapter 1

Introduction

The TMS320 family of digital signal processors (DSPs) combines the high performance required in DSP applications with special features for these applications.

A complete set of hardware and software development tools complement The TMS320C3x/C4x DSPs. These include an optimizing C compiler, an assembler, a linker, an archiver, a software simulator, a full-speed emulator, and a software development board. Section 1.1 describes these tools.

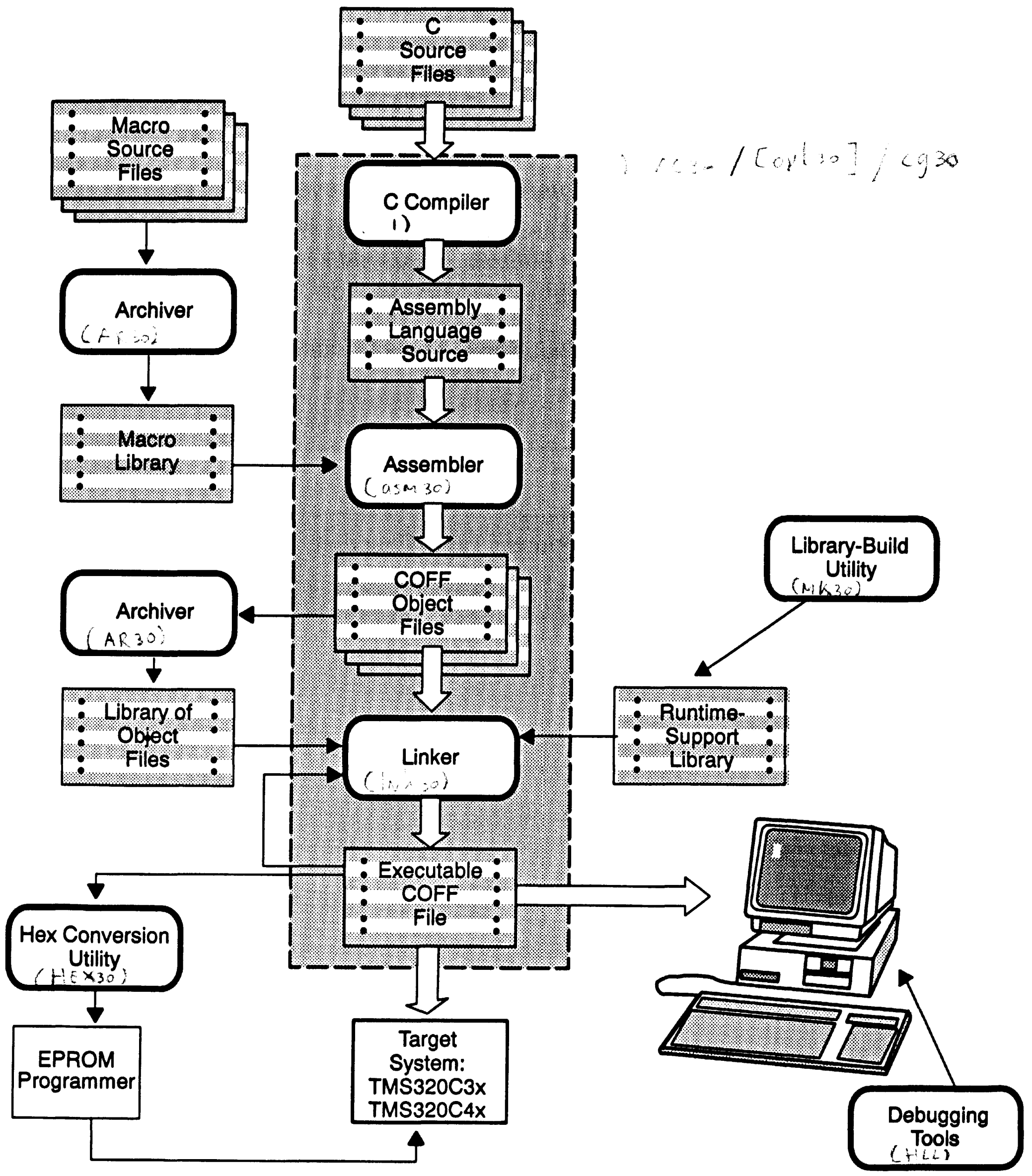
These are the topics included in this introductory chapter:

Topic **Page**

1.1 Software Development Tools Overview

Figure 1–1 illustrates the TMS320 floating-point software development flow. The shaded portion of the figure highlights the most common path of software development; the other portions are optional.

Figure 1–1. TMS320C3x/C4x Software Development Flow



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

- The **C compiler** accepts C source code and produces TMS320C3x or TMS320C4x assembly language source code. A **shell program (cl30)**, an **optimizer (opt30)**, and an **interlist utility (clist)** are included in the compiler package. The shell program allows you to automatically compile, assemble, and link source modules. The interlist utility shows you original C source statements with their assembly language output. Chapter 2 describes how to invoke and operate the compiler, the shell, the optimizer, and the interlist utility.
- The **assembler** translates assembly language source files into machine language COFF object files.
- The **archiver (ar30)** allows you to collect a group of files into a single archive file, called a *library*. It also allows you to modify a library by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is to build a library of object modules.
- Two object libraries are shipped with the C compiler:
  - **rts30.lib** contains ANSI-standard runtime-support functions and compiler-utility functions for the TMS320C3x small memory model with the standard runtime model.
  - **rts40.lib** contains ANSI-standard runtime-support functions and compiler-utility functions for the TMS320C4x small memory model with the standard runtime model.
- Use the **library-build utility (mk30)**, to build your own customized runtime-support library. Standard runtime-support library functions are provided as source code located in rts30.src and rts40.src.
  - **rts30.src** contains standard runtime functions for the TMS320C3x processors.
  - **rts40.src** contains standard runtime functions for the TMS320C4x processors.
  - **mathasm.src** contains assembly language source for trigonometric functions
  - **prts40.src** contains C and assembly language routines for handlihg peripherals and interrupts for 'C4x devices
  - **prts30.src** contains C and assembly language routines for handling peripherals/interrupts for 'C3x devices
- The **linker (lnk30)** 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.

- The **hex conversion utility (hex30)** converts a COFF object file into an Intel, Tektronix, or TI-Tagged object format file that you can download to an EPROM programmer.
- The main purpose of this development process is to produce a module that can be executed in a **TMS320C3x/C4x target system**. You can use one of several debugging tools to refine and correct your code before downloading it to a TMS320C3x/C4x target system. These debugging platforms share a common screen-oriented interface that allows you to display machine status information, inspect and modify C variables, display C source code, and monitor the execution of your program as it runs on the debugging platform. Available debugging tools include:
  - An instruction-accurate software **simulator** that simulates the TMS320C3x/C4x functions. The simulator executes linked COFF object modules.
  - An **XDS (extended development system) emulator**, which is a PC-resident, real-time, in-circuit emulator that features the same screen-oriented interface as the simulator
  - An **EVM (evaluation module),** which is a plug-in PC board that contains a target CPU, such as a C30, that can be used to evaluate CPU performance

1.2 TMS320 Floating-Point C Compiler Overview

The TMS320 floating-point C compiler is a full-featured optimizing compiler that translates standard ANSI C programs into TMS320C3x/C4x assembly language source. The following list describes key features of the compiler:

ANSI Standard C

The TMS320 floating-point compiler fully conforms to the ANSI C standard as defined by the ANSI specification and described in Kernighan and Ritchie's *The C Programming Language* (second edition). The ANSI standard includes recent extensions to C that are now standard features of the language. These extensions provide maximum portability and increased capability.

Optimization

The compiler uses a sophisticated optimization pass that employs several advanced techniques for generating efficient, compact code from C source. General optimizations can be applied to any C code, and TMS320C3x/C4x-specific optimizations take advantage of the particular features of the TMS320C3x/C4x architecture. For more information about the C compiler's optimization techniques, refer to Section NO TAG on page NO TAG and to Appendix A.

ANSI Standard Runtime Support

The compiler package comes with two complete runtime libraries (rts30.lib and rts40.lib). All library functions conform to the ANSI C library standard. The libraries include functions for string manipulation, dynamic memory allocation, data conversion, timekeeping, and trigonometry, plus exponential and hyperbolic functions. Functions for I/O and signal handling are not included, because these are target-system specific. For more information, refer to Chapter 5.

Assembly Source Output

The compiler generates assembly language source that is easily inspected, enabling you to see the code generated from the C source files.

Big and Small Memory Models

The compiler supports two memory models. The small memory model enables the compiler to efficiently access memory by restricting the global data space to a single 64K-word data page. The big memory model allows unlimited data space. For more information, refer to subsection 2.3.1 on page 2-31.

Compiler Shell Program

The compiler package includes a shell program, which enables you to compile, assemble, and link programs in a single step. For more information, refer to Section 2.1 on page 2-2.

Flexible Assembly Language Interface

The compiler has straightforward calling conventions, allowing you to easily write assembly and C functions that call each other. For more information, refer to Chapter 4, *Runtime Environment*.

Integrated Preprocessor

The C preprocessor is integrated with the parser, allowing for faster compilation. Standalone preprocessing or preprocessed listing is also available. For more information, refer to Section 2.2 on page 2-26.

COFF Object Files

Common object file ~~f~~ F ormat (COFF) allows you to define your system's memory map at link time. This maximizes performance by enabling you to link C code and data objects into specific memory areas. COFF also provides rich support for source-level debugging.

ROM-able Code

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

Source Interlist Utility

The compiler package includes a utility (clist) that interlists your original C source statements into the assembly language output of the compiler. This utility provides you with an easy method for inspecting the assembly code generated for each C statement. For more information, refer to Section 2.6 on page 2-44.

32-Bit Data Sizes

All data sizes (char, short, int, long, float, and double) are 32 bits. This allows all types of data to take full advantage of the TMS320C3x/C4x's 32-bit integer and floating-point arithmetic capabilities. For more information, refer to Section 3.2 on page 3-4.

Library-Build Utility

A library-build utility called *mk30* allows you to easily custom-build object libraries from source for any combination of runtime models or target CPUs.

Chapter 2

C Compiler Description

Translating your source program into code that the TMS320C3x/C4x can execute is a process consisting of several steps. You must compile, assemble, and link your source files to create an executable object file. The TMS320 floating-point package contains a special cl30 shell program, that enables you to execute all of these steps with one command. This chapter provides a complete description of how to use the shell program to compile, assemble, and link your programs.

The TMS320 floating-point C compiler includes an optimizer that allows you to produce highly optimized code. The optimizer is explained in Section 2.4.

The compiler package also includes a utility that interlists your original C source statements into the compiler's assembly language output. This enables you to inspect the assembly language code generated for each C statement. The interlist utility is explained in Section 2.6.

This chapter includes the following topics:

Topic **Page**

2.1 Compiling C Code

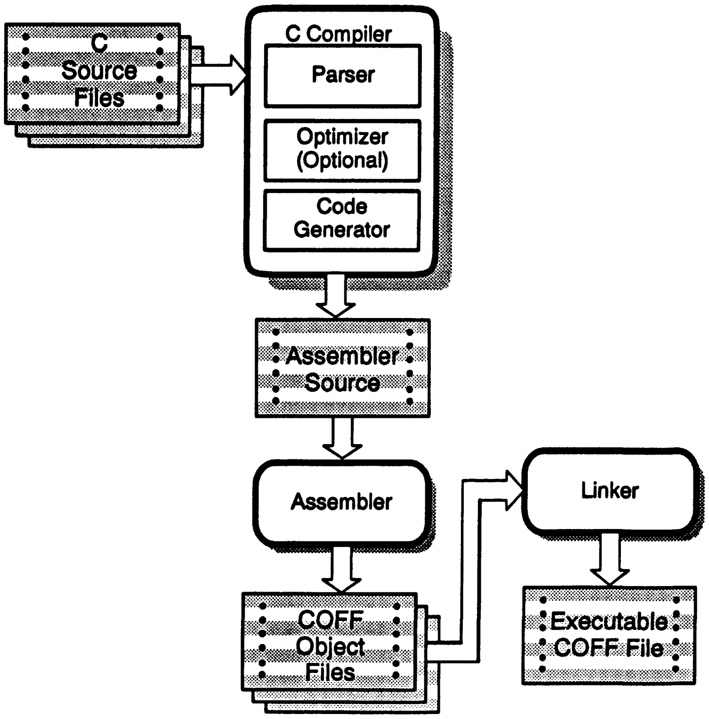
The cl30 shell program is a utility that lets you compile, assemble, and optionally link in one step. The shell runs one or more source modules through the following:

The **compiler**	which includes the parser, the optimizer, and the code generator.
The **assembler**	which generates a COFF object file.
The **linker** (optional)	which links your files to create an executable object file. The linker can be invoked as part of the larger process, or you can compile and assemble various files with the shell and link at a later time.

For more information about the floating-point assembler and linker, refer to the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide.*

By default, the shell compiles and assembles files; however, if you use the –z option, the shell also links your files. Figure 2–1 illustrates the path the shell takes with and without the –z option.

Figure 2–1. The cl30 Shell Program Overview



2.1.1 Invoking the C Compiler

To run the compiler, enter:

```
cl30 [–options] [filenames] [–z [link_options] [object files]]
```

cl30	is the command that invokes the compiler and the assembler.
–options	affect the way the compiler processes input files.
filenames	are one or more C source files, assembly source files, or object files.
–z	is the option that runs the linker.
link_options	control the linking process.
object files	name the object files that the compiler creates.

Options control how the compiler processes files, and the filenames provide a method of identifying source files, intermediate files, and output files. Options and filenames can be specified in any order on the command line. However, if you use the –z option, this option and its associated information must follow all filenames and compiler options. For example, if you wanted to compile two files named *symtab* and *file*, assemble a third file named *seek.asm*, and use the quiet option (–q), you would enter:

```
cl30 –q symtab file seek.asm
```

As cl30 encounters each source file, it prints the filename in square brackets [for C files] or angle brackets <for asm files>. The example above uses the –q option to suppress the additional progress information that cl30 produces. Entering the command above produces:

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

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

```
$ cl30 symtab
[symtab]
TMS320C3x/4x ANSI C Compiler        Version x.xx
Copyright (c) 1987–1995, Texas Instruments Incorporated
   "symtab.c":==>  main
   "symtab.c":==>  lookup
TMS320C3x/4x ANSI C Codegen         Version x.xx
Copyright (c) 1987–1995, Texas Instruments Incorporated
   "symtab.c":==>  main
   "symtab.c":==>  lookup
TMS320c3x/4x COFF Assembler         Version x.xx
Copyright (c) 1987–1995, Texas Instruments Incorporated
   PASS 1
   PASS 2

No Errors, No Warnings
```

2.1.2 Specifying Filenames

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

Extension	File Type
.asm, **.abs**, or **.s*** (extension begins with s)	assembly language source file
.c or **no extension**	C source file
.o* (extension begins with o)	object file

Files without extensions are assumed to be C source files, and a .c extension is assumed.

You can use the –e option to change these default extensions, causing the shell to associate different extensions with assembly source files or object files. You can also use the –f option on the command line to override these file type interpretations for individual files. For more information about the –e and –f options, refer to page 2-12.

The conventions for filename extensions allow you to compile C files and assemble assembly files with a single command, as shown in the example on page 2-3.

You can use wildcard filename specifications to compile multiple files. Wildcard specifications vary by system; use the appropriate form.

To compile all the files in a directory that have a .c extension (by default, all C files), enter the following (DOS system):

```
cl30 *.c
```

2.1.3 Compiler Options

Command line options control the operation of both the shell and the programs it runs. This section provides a description of option conventions, an option summary table, and a detailed description of each of the options.

Compiler options:

- Consist of single letters or two-letter pairs
- Are not case sensitive
- Are preceded by a hyphen
- Can be combined if there are single-letter options without parameters: for example, –sgq is equivalent to –s –g –q.
- Can be combined if two-letter pair options without parameters have the same first letter: for example, –mr and –mb can be combined as –mrb.
- Cannot be combined with other options if they have parameters, such as –u*name* and –i*dir.*

You can set up default options for the shell by using the C_OPTION environment variable. For a detailed description of the C_OPTION environment variable, refer to subsection 2.1.4, *Using the C_OPTION Environment Variable,* on page 2-23.

Table 2–1 summarizes all compiler options. The table is followed by in-depth descriptions of each of the options.

Table 2–1. Compiler Options Summary Table

General Shell Options	Option	Effect
These options control the overall operation of the cl30 shell. For more information, see page 2-10.	–c	no linking (negates –z)
	–d*name*[=def]	predefine a constant
	–g	enable symbolic debugging
	–i*dir*	define #include search path
	–k	keep .asm file
	–n	compile only (create .asm file)
	–q	suppress progress messages (quiet)
	–qq	suppress all messages (super quiet)
	–s	interlist C and asm source statements
	–u*name*	undefine a constant
	–v*xx*	determine processor; *xx* = 30 or 40
	–z	enable linking
File Specifiers	**Option**	**Effect**
These options use the extensions of each filename to determine how to process the file. For more information, see page 2-12.	–ea	set default extension for assembly files
	–eo	set default extension for object files
	–fa *file*	identify assembly language file (default for .asm or .s*)
	–fc *file*	identify C source file (default for .c or no extension)
	–fo *file*	identify object file (default for .o*)
	–fr*dir*	specify object file directory
	–fs*dir*	specify assembly file directory
	–ft	override TMP environment variable

Table 2–1. Compiler Options Summary Table (Continued)

Parser Options	**Option**	**Effect**
These options control the preprocessing, syntax-checking, and error-handling behavior of the compiler. For more information, see page 2-14.	–pe	treat code-E errors as warnings
	–pf	generate prototypes for functions
	–pk	allow compatibility with pre-ANSI K&R C
	–pl	generate preprocessed listing (.pp) file
	–pn	suppress #line directives in .pp file
	–po	preprocess only
	–pw	suppress warning messages
	–p?	enable trigraph expansion
Inlining Options	**Options**	**Effect**
These options control expansion of functions declared as inline. For more information, refer to page 2-15.	–x*n*	(expand inline functions; *n* tells the compiler which functions to inline.)
	–x0	0 disable inlining
	–x1	1 default inlining level
	–x2	2 define _INLINE + invoke optimizer at level 2
Type-Checking Options	**Option**	**Effect**
These options allow relaxation of compiler type-checking rules. For more information, refer to page 2-16.	–tf	relax prototype checking
	–tp	relax pointer combination checking
Runtime Model Options	**Options**	**Effect**
These options are used to customize the executable output of the compiler for your specific application. For more information, see page 2-17.	–ma	assume aliased variables
	–mb	select big memory configuration
	–mc	allow faster float to int conversions
	–mf	force indirect access to external objects
	–mi	disable RPTS instructions for loops
	–mm	enable short multiply ('C3x only)
	–mn	enable optimizer options disabled by –g
	–mr	use register argument model
	–mx	support first pass TMX silicon

Table 2–1. Compiler Options Summary Table (Continued)

Assembler Options	Options	Effect
These options control the assembler's behavior. For more information, see page 2-18.	–aa	create absolute listing
	–al	produce assembly listing file
	–as	keep labels as symbols
	–ax	produce cross-reference file
Linker Options	**Options**	**Effect**
These options are valid only when the compiler has been invoked with the –z option for linking. They control the linking process. They must follow the –z option on the command line, and the –z option must follow all other options and file-names on the command line. For more information, see page 2-19.	–a	generate absolute output
	–ar	generate relocatable output
	–b	disable the merging of symbolic debug information
	–c	use ROM initialization
	–cr	use RAM initialization
	–e *sym*	define entry point
	–f*val*	define fill value
	–g	make all global symbols static regardless of the use of –h
	–h	make all global symbols static
	–heap *size*	set heap size (words)
	–i *dir*	define library search path
	–l *lib*	supply library name
	–m *file*	name the map file
	–n	ignore all fill specifications in memory directives
	–o *file*	name the output file
	–r	generate relocatable output
	–s	strip symbol table
	–stack *size*	set stack size (bytes)
	–u *sym*	undefine entry point
	–x	force rereading of libraries

Table 2–1. Compiler Options Summary Table (Continued)

Optimizer Options	Options	Effect
These options control the behavior of the optimizer. For more information, see page 2-21.	–o0	perform level 0 (register) optimization
	–o1	perform level 1 (level 0+ local) optimization
	–o2 (or –o)	perform level 2 (level 1+ global) optimization
	–o3	perform level 3 (level 2+ file) optimization
	~~–oe~~	~~specify that generated code is not reentrant~~
	–oi*size*	set automatic inlining size (–o3 only)
	–olO (–oLO)	specify that this file alters a standard library function
	–ol1 (–oL1)	specifies that this file defines a standard library function
	–ol2 (–oL2)	specify that this file does not define or alter library functions
	–on0	disable optimizer information file
	–on1	produce optimizer information file
	–on2	produce a verbose information file
	–opO	specify that callable functions and/or modifiable varables are used in this module
	–op1	specify that no callable functions are used in this module (default)
	–op2	specify that no modifiable varaibales or callable functions are used in this module
	–ou	allow zero-overhead loop operations

General Shell Options

You can use the options described below to control the overall operation of the cl30 shell.

–c	suppresses the linking option; it causes the shell not to run the linker even if –z is specified. This option is especially useful when you have –z specified in the C_OPTION environment variable and you don't want to link. For more information, refer to page 2-56.
–d*name*	–d*name def* predefines a constant for the preprocessor. This is equivalent to inserting *#define name def* at the top of each C source file. If the optional *def* is omitted, *–dname* sets *name* equal to 1.
–g	causes the compiler to generate symbolic debugging directives that are used by the C source level debuggers.
–i*dir*	adds *dir* to the list of directories to be searched for #include files. You can use this option a maximum of 32 times to define several directories; be sure to separate –i options with spaces. Note that if you don't specify a directory name, the preprocessor ignores the –i option. For more information, refer to subsection 2.2.2 on page 2-26.
–k	keeps the assembly language file. Normally, the shell deletes the output assembly language file after assembling completes, but using –k allows you to retain the assembly language output from the compiler.
–n	causes the shell to compile only. If you use –n, the specified source files are compiled but not assembled or linked. This option overrides –z and –c. The output of –n is assembly language output from the compiler.
–q	suppresses banners and progress information from all the tools. Only source filenames and error messages are output.
–qq	suppresses *all* output except error messages.

–s	invokes the interlist utility, which interlists C source statements into the compiler's assembly language output. This allows you to inspect the code generated for each C statement. This option automatically uses the –k option. For more information, see Section 2.6 , page 2-43.
–u*name*	undefines the predefined constant *name*. Overrides any –d options for the specified constant.
–v*xx*	specifies the target processor (*xx*). Choices are 30 for a 'C3x processor, or 40 for a 'C4x. By default, the tools produce code for the TMS320C3x. Use **–v40** to generate code for the TMS320C4x. All code used in the eventual executable module, including all library-resident code, must be compiled under the same version. For more information, refer to Section 2.3, page 2-29.
–z	enables the linking option. It causes the shell to run the linker on specified object files. The –z option and its parameters follow all other compiler options and source files on the command line. All arguments that follow –z on the command line are passed to, and interpreted by, the linker. For more information, refer to subsection 2.9.1 on page 2-54.

File Specifiers

–e allows you to change the default naming conventions for file extensions for assembly language files and object files. This affects the interpretation of source filenames as well as the naming of files that the shell creates.

The syntax for the –e option is:

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

For example:

```
cl30 -ea .rrr -eo .odsp fit.rrr
```

assembles the file fit.rrr and creates an object file named fit.odsp.

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

```
cl30 -earrr -eoodsp fit.rrr
```

The –e option should precede any filenames on the command line. If you don't use the –e option, the default extensions are .asm for assembly files and .obj for object files.

–f	overrides default interpretations for source file extensions. If your naming conventions do not conform to those of the shell, you can use –f options to specify which files are C source files, assembly files, or object files. You can insert an optional space between the –f option and the filename.

The –f options are:

–fa*file*	for assembly source file
–fc*file*	for C source file
–fo*file*	for object file

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

```
cl30 –fcfile.s –fa assy
```

Note that –f cannot be applied to a wildcard specification.

–fr permits you to specify a directory for object files. If the –fr option is not specified, the shell will place object files in the current directory. To specify an object file directory, insert the directory's pathname on the command line after the –fr option:

```
cl30 –fr d:\object ...
```

–fs permits you to specify a directory for assembly files. If the –fs option is not specified, the shell will place assembly files in the current directory. To specify an assembly file directory, insert the directory's pathname on the command line after the –fs option:

```
cl500 –fs d:\assembly ...
```

–ft permits you to specify a directory for temporary intermediate files. The –ft option overrides the TMP environment variable (described in subsection 2.1.5). To specify a temporary directory, insert the directory's pathname on the command line after the –ft option:

```
cl30 –ft d:\temp ...
```

Parser Options

–pe	treats code-E errors as warnings. Normally, the code generator does not run if the parser detects any code-E errors. When you use the –pe option, the parser treats code-E errors as warnings, allowing complete compilation. For more information about errors and about –pe, refer to Section 2.7 on page 2-44.
–pf	produces a function prototype listing file. The parser creates a file containing the prototype of every procedure in all corresponding C files. Each function prototype file is named as its corresponding C file, with a .pro extension. –pf is useful when conforming code to the ANSI C standard, or generating a listing of procedures defined.
–pk	relaxes certain requirements that are stricter than those required by earlier K&R compilers, and that are newly imposed by the ANSI C standard. This facilitates compatibility between existing K&R-compatible programs and the TMS320C3x ANSI compiler. The effects of the –pk options are described in Section 3.7 on page 3-10.
–pl	generates a preprocessed listing file. The compiler writes a modified version of the source file to an output file called file.pp. This file contains all the source from #include files and expanded macros. It does not contain any comments. For more information, refer to subsection 2.2.3 on page 2-28.
–pn	suppresses line and file information. –pn causes #line directives of the form: `#123 file.c.` to be suppressed in a file generated with –po or –pl. You may find –pn useful when compiling machine-generated source.
–po	runs the compiler for preprocessing only. When invoked with –po, the compiler processes only macro expansions, #include files, and conditional compilation. The compiler writes the preprocessed file with a .pp extension. For more information, refer to subsection 2.2.3 on page 2-28.

–pw suppresses warning messages (code-W errors). The compiler produces diagnostic messages for only those errors that prevent complete compilation. This option can be doubled (–pww) to suppress code-E error messages also. For more information, refer to Section 2.7 on page 2-44.

–p? enables trigraph expansion. Trigraphs are special escape sequences of the form:

```
??c
```

where *c* is a character. The ANSI C standard defines these sequences for the purpose of compiling programs on systems with limited character sets. By default, the compiler does not recognize trigraphs; use –p? to enable trigraphs. For more information, refer to the ANSI specification, subsection 2.2.1.1. or K&R § A 12.1.

Inlining Options

–x*n* controls function inlining done by the optimizer when functions have been defined or declared as *inline*. The possibilities are:

–x0:	disables all inlining,
–x1:	inlines all intrinsic operators, and
–x2: (or–x)	invokes the optimizer at level 2 and defines the _INLINE preprocessor symbol, which causes all functions defined or declared as inline to be expanded in line.

Note that –x1 is the default inlining option. It occurs whether or not the optimizer is invoked and whether or not any –x options are specified. The last option may be specified as –x or –x2 interchangeably. See Section 2.5 on page 2-37 for more details.

Type-Checking Options

–tf relaxes type checking on redeclarations of prototyped functions. In ANSI C, if a function is declared with an old-format declaration, such as:

```
int func( );
```

and then later declared with a prototype, such as:

```
int func(float a, char b);
```

this generates an error because the parameter types in the prototype disagree with the default argument conversions (which convert float to double and char to int). With the –tf option, the compiler overlooks such redeclarations of parameter lists.

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

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

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

- Pointers to differently qualified types can be combined:

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

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

Runtime-Model Options

–ma assumes that variables are aliased. The compiler assumes that pointers may alias (point to) named variables and therefore aborts register optimizations when an assignment is made through a pointer.

–mb selects the big memory model, allowing unlimited space for global data, static data, and constants. In the small memory model, which is the default, this space is limited to 64K words. All code used in the eventual executable module, including all library resident code, must be compiled under the same model. For more information, refer to Section 2.3, page 2-29.

–mc allows faster float-to-int conversion. The ANSI C standard specifies that when an object of floating-point type is converted to an integer type, the fractional part is discarded, effectively rounding towards zero. The compiler uses the TMS320C3x/C4x FIX instruction for these conversions, which rounds toward negative infinity, followed by a four-instruction sequence to correct negative values. If the ANSI standard behavior is not important to your application, the –mc option suppresses the correction sequence for faster execution.

–mf forces the compiler to always honor indirection on external objects. The compiler allows objects not declared in the .bss section to be accessed indirectly via pointers, as described in subsection 4.5.2 on page 4-24. Such objects cannot be accessed directly because they are not on the data page addressed by the DP. In some cases the optimizer may convert such indirect accesses into direct accesses, possibly resulting in objects in the .bss being accessed incorrectly. For example:

```
extern struct sss ext_obj;/*object not in .bss*/
struct sss ext_ptr = &ext_obj /*object pointer*/
```

The optimizer may convert an expression such as structure member ext_ptr–>f1 to ext_obj.f1 because the optimizer assumes that all variables are in the .bss section. This transformation is invalid if ext_obj is not in the .bss section. The –mf option inhibits the transformation and preserves the indirection.

–mi disables the use of RPTS instructions for loops and uses RPTB instead. This allows loops to be interrupted.

–mm	(TMS320C3x only) enables short multiplies, generating MPYI instructions for integer multiplies rather than runtime support calls. If your application does not need 32-bit integer multiplication, use –mm to enable the MPYI instruction; it is significantly faster because it performs 24x24 bit multiplication. For more information, refer to Section 4.7 on page 4-31.
–mn	re-enables the optimizations disabled by –g. If you use the –g option to generate symbolic debugging information, many code generator optimizations are disabled because they disrupt the debugger.
–mr	uses the register-argument model. Code compiled under this model is completely incompatible with code compiled under the default model. *All code used in the eventual executable module, including all library resident code, must be compiled under the same model.* For more information, refer to Section 2.3 on page 2-29.
–mx	supports first-pass TMX silicon. The –mx option enables the code generator to work around some of the known hardware bugs in early TMS320C3x/C4x devices.

Assembler Options

–aa	invokes the assembler with the –a option, which creates an absolute listing. An absolute listing shows the absolute addresses of object code.
–al	invokes the assembler with the –l (lowercase L) option to produce an assembly listing file.
–as	retains labels. Label definitions are written to the COFF symbol table for use with symbolic debugging.
–ax	invokes the assembler with the –x option to produce a symbolic cross-reference in the listing file.

For more information about assembler options, see the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide.*

Linker Options

Linker options can be used with the compiler, or with the linker as a standalone. (See Section 2.9.2, *Linking C Code,* on page 2-55). When used with the compiler shell, all linker options should follow the –z option described in *General Options*, on page 2-10. For example:

```
cl30  –q symtab.c –z –a –c –o symtab.out –l rts30.lib
```

In this example, the file symtab.c will be compiled with the –q (quiet) option. The –z option causes the shell to invoke the linker and pass the –a, –c, –o, and –l linker options to the linker.

All compiler command line options following –z are passed to the linker. For this reason, the –z option followed by the linker options must be the last shell option specified on the command line. All options on the command line following the –z option will be passed to the linker and not the compiler.

The –c and –n options suppress the linker option and cause the shell not to run the linker even if –z has been specified (see *General Shell Options*, page 2-10, for more information.) Linker Options are summarized in Table 2–1, *Options Summary Table*, beginning on page 2-6. For more information about linker options, refer to Section 2.9, page 2-54.

For more information, see Chapter 8 of *The TMS320 Floating-Point DSP Assembly Language Tools.*

–a	produces an absolute, executable module. This is the default; if neither –a nor –r is specified, the linker acts as if –a is specified.
–ar	produces a relocatable, executable object module.
–b	disables merging of symbolic debugging information.
–c	enables linking conventions defined by the ROM autoinitialization model of the TMS320C3x/4x C compiler.
–cr	enables linking conventions defined by the RAM autoinitialization model of the TMS320C23x/4x C compiler.
–e *global_symbol*	defines a *global_symbol* that specifies the primary entry point for the output module.
–f *fill_value*	sets the default fill value for holes within output sections; *fill_value* is a 16-bit constant.
–h	makes all global symbols static.
–heap *size*	sets heap size (for the dynamic memory allocation in C) to *size* words and defines a global symbol that specifies the heap size. Default = 1K words.

–i *dir*	alters the library-search algorithm to look in *dir* before looking in the default location. This option must appear before the –l option. The directory must follow operating system conventions.
–l *filename*	names an archive library file as linker input; *filename* is an archive library name, and must follow operating system conventions.
–m *filename*	produces a map or listing of the input and output sections, including holes, and places the listing in *filename*. The *filename* must follow operating system conventions.
–o *filename*	names the executable output module. The default *filename* is a.out, and must follow operating system conventions.
–q	suppresses banner and progress information.
–r	retains relocation entries in the output module.
–s	strips symbol table information and line number entries from the output module.
–stack *size*	sets the C system stack size to *size* words and defines a global symbol that specifies the stack size. Default = 1K words.
–u *symbol*	places the unresolved external symbol *symbol* into the output module's symbol table.
–v0	generates version 0 COFF format.
–w	generates a warning when an output section that is not specified with the SECTIONS directive is created.
–x	forces rereading of libraries, and resolves back references.

Optimizer Options

–o*n* causes the compiler to optimize the intermediate file that is produced by the parser. *n* denotes the level of optimization. There are three levels of optimizations: **–o0**, **–o1**, **–o2**, and **–o3**.

If you do not indicate a level (0, 1, 2, 3) after the –o option, the optimizer defaults to level 2. For more information about the optimizer, refer to Section 2.4 on page 2-32 and Appendix A.

–oe allows the optimizer to generate code that is not reentrant. When the –oe option is used, the optimizer will place variables declared with the auto storage class specifier into static storage instead of the stack if they cannot be allocated to registers and are not live across function calls. The –ms option implies the –oe option.

–oi*size* controls automatic inlining of functions (not defined or declared as inline) at optimization level 3. You specify the *size* limit for the largest function that will be inlined (times the number of times it is called). If no size is specified, the optimizer will inline only very small functions. Setting the size to 0 (–oi0) disables automatic inlining completely. Note that this option controls only the inlining of functions that have not been explicitly defined or declared as inline. The –x options control the inlining of functions declared as inline.

RROR.
ting Stated
ule, 2.4

–ol*n* (lowercase L) controls file level optimizations. When you invoke the optimizer at level 3 (–o3), some of the optimizations use known properties of the standard library functions. If your file redefines any of these standard functions, these optimizations become ineffective, or the compiler may produce unintentional errors. Use the –ol option to notify the optimizer if any of the following situations exist:

–ol0 :	defines a function with the same name as a standard library function.
–ol1 :	contains the standard library definition functions.
–ol2 :	does not alter standard library functions. Restores the default behavior of the optimizer if you have used one of the other two options in a command file, an environment variable, etc.

–on*n* causes the compiler to produce a user readable optimization information file with a .nfo extension. This option works only when the –o3 option is used. There are three levels available:

–on0 :	Do not produce an information file. Restores the default behavior of the optimizer if you have used one of the other two options in a command file, an environment variable, etc.
–on1 :	Produce an optimization information file.
–on2 :	Produce a verbose optimization information file.

–op*n* specifies whether functions in other files can call this file's EXTERN functions, or modify this file's EXTERN variables. Level 3 optimization combines this information with its own file-level analysis to decide whether to treat this file's EXTERN function and variable definitions as if they had been declared STATIC. The following three levels are defined.

–op0 :	Signals the optimizer that functions in this module may be called by other modules, and variables declared within the module may be altered by other modules. This disables some of the –o3 optimizations.
–op1 :	Signals the optimizer that no functions in this module will be called by other modules and no interrupt functions declared elsewhere will call functions defined in this module. This is the default when –o3 is used.
–op2/ or –op :	Signals the optimizer that no functions in this module are called by other modules and no variable declared in this module will be altered by another module.

–ou allows the compiler to use zero-overhead loop instructions, RPTS and RPTB to control unsigned loop counters. To use this option, you must be certain that these loops will iterate fewer than 2^{31} times.

2.1.4 Using the C_OPTION Environment Variable

An environment variable is a system symbol that you define and assign to a string. You may find it useful to set the shell default options using the C_OPTION environment variable; if you do this, these default options and/or input filenames are used every time you run the shell.

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

Options specified with the environment variable 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, enable symbolic debugging, and link, then set up the C_OPTION environment variable as follows:

Host	Enter
DOS or OS/2	set C_OPTION=−qg −z
UNIX	setenv C_OPTION "−qg −z"

You may want to set C_OPTION in your system initialization file; for example, on PCs, in your autoexec.bat file.

Using the −z option enables linking. If you plan to link most of the time when using the shell, you can specify the −z option with C_OPTION. Later, if you need to invoke the shell without linking, you can use −c on the command line to override the −z specified with C_OPTION. These examples assume C_OPTION is set as shown previously:

```
cl30 *.c              ; compiles and links
cl30 −c *.c           ; only compiles
cl30 *.c −z c.cmd     ; compiles/links using command file
cl30 −c *.c −z c.cmd  ; only compiles (−c overrides −z)
```

2.1.5 Using the TMP Environment Variable

The shell program creates intermediate files as it processes your program. For example, the parser phase of the shell creates a temporary file used as input by the code generation phase. By default, the shell puts intermediate files in the current directory. However, you can name a specific directory for temporary files.

This feature allows use of a RAM disk or other high-speed storage files. It also allows source files to be compiled from a remote directory without writing any files into the directory where the source resides. This is useful for protected directories.

There are two ways to specify a temporary directory:

1) Use the TMP environment variable:

   ```
   set TMP=d:\temp
   ```

   This example is for a PC. Use the appropriate command for your host.

2) Use the –ft option on the command line:

   ```
   cl30 –ft d:\temp....
   ```

   The –ft option, if used, overrides the TMP environment variable.

2.2 Controlling the Preprocessor

The TMS320 floating-point C compiler includes standard C preprocessing functions, which are built into the first pass of the compiler (parser). 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)

This section describes specific features of the TMS320C3x/C4x preprocessor. A general description of C preprocessing is in Section A12 of K&R.

2.2.1 Predefined Names

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

Table 2–2. Predefined Macro Names

Macro Name	Description
__LINE__ †	expands to the current line number
__FILE__ †	expands to the current source filename
__DATE__ †	expands to the compilation date, in the form *mm dd yyyy*
__TIME__ †	expands to the compilation time, in the form *hh:mm:ss*
__STDC__ †	expands to 1 (identifies the compiler as ANSI standard)
_TMS320C3x	expands to 1 if the target processor is a TMS320C3x processor, otherwise it is undefined.
_TMS320C4x	expands to 1 if the target processor is a TMS320C4x processor, otherwise it is undefined.
_INLINE	expands to 1 under the –x or –x2 optimizer option, undefined otherwise
_REGPARM	expands to 1 if the register argument runtime model is used, undefined otherwise.
_BIGMODEL	expands to 1 if the –mb option is used, undefined otherwise.

† Specified by the ANSI standard

You can use these names in the same manner as any other defined name. For example:

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

could translate to a line such as:

```
printf (" %s %s", "Jan 14 1994", "13:58"17");
```

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

You can predefine additional names from the command line by using the –d option:

```
cl30 –dNAME –dREGS=6 *.c
```

This has the same effect as including these lines at the beginning of each source file:

```
#define NAME 1
#define REGS 6
```

2.2.2 #include File Search Paths

The #include preprocessor directive tells the compiler to read source statements from another file. The syntax for this directive is:

```
#include "filename"
        or
#include <filename>
```

The *filename* names the #include file that the compiler reads statements from. You can enclose the filename in double quotes or in angle brackets. The filename can be a complete pathname, have partial path information, or have no path information.

- If you enclose the filename in *double quotes*, the compiler searches for the file in the following directories, in the order given:
  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) Any directories named with the –i compiler option in the shell.
  3) Any directories set with the environment variable C_DIR.
- If you enclose the filename in *angle brackets*, the compiler searches for the file in the following directories, in the order given:
  1) Any directories named with the –i option in the shell.
  2) Any directories set with the environment variable C_DIR.

  Note that if you enclose the filename in angle brackets, the compiler *does not* search for the file in the current directory.

Include files are sometimes stored in directories. You can augment the compiler's directory search algorithm by using the –i shell option or the environment variable C_DIR to identify a directory name.

–i Shell Option

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

```
cl30 –i pathname ...
```

You can use up to 32 –i options per invocation; each –i option names one *pathname*. In C source, you can use the #include directive without specifying any path information for the file; instead, you can specify the path information with the –i option. For example, assume that a file called source.c is in the current directory. This file contains one of the following directive statements:

```
#include "alt.h" or
#include <alt.h>
```

The table below lists the complete pathname for alt.c and shows how to invoke the compiler; select the row for your host system.

Host	Pathname for alt.c	Invocation Command
DOS or OS/2	c:\dsp\files\alt.h	cl30 –ic:\dsp\files source.c
UNIX	/dsp/files/alt.h	cl30 –i/dsp/files source.c

C_DIR Environment Variable

The compiler uses the environment variable **C_DIR** to name alternate directories that contain #include files. To specify the same directory for #include files, as in the previous example, set C_DIR with one of these commands:

Host	Enter
DOS or OS/2	set C_DIR=c:\dsp\files
UNIX	setenv C_DIR "/dsp/files"

Then you can include alt.h:

```
#include "alt.h" or
#include <alt.h>
```

and invoke the compiler without the –i option:

```
cl30 source.c
```

This results in the compiler using the path in the environment variable to find the #include file.

The pathnames specified with C_DIR are directories that contain #include files. You can separate pathnames with a semicolon or with blanks. In C source, you can use the #include directive without specifying any path information; instead, you can specify the path information with C_DIR.

The environment variable remains set until you reboot the system or reset the variable by entering one of these commands:

Host	Enter	
DOS or OS/2	set	C_DIR=
UNIX	unsetenv	C_DIR

2.2.3 Generating a Preprocessed Listing File (–pl, –pn, –po Options)

The –pl shell option allows you to generate a preprocessed version of your source file. The compiler's preprocessing functions perform the following actions on the source file:

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

(These functions correspond to translation phases 1–3 as specified in Section A12 of K&R.)

The preprocessed output file contains no preprocessor directives other than #line; the compiler inserts #line directives to synchronize line and file information in the output files with input position from the original source files. If you use the –pn option, no #line directives are inserted.

If you use the –po option, the compiler performs *only* the preprocessing functions listed above and then writes out the preprocessed listing file; no syntax checking or code generation takes place. The –po option can be useful when debugging macro definitions or when host memory limitations dictate separate preprocessing (refer to *Parsing in Two Passes* on page 2-49). The resulting preprocessed listing file is a valid C source file that can be rerun through the compiler.

2.3 Using Runtime Models

The compiler has three options that allow you to affect the runtime model. All linked modules must use the same runtime model in order to correctly interface with other C modules, assembly modules, and library resident modules. Runtime models are mutually exclusive, that is, you must choose a configuration and then make sure that all the modules you link use the same model. The following conventions are mutually exclusive and must be chosen before you begin to compile:

Mutually Exclusive Runtime Model Options	
big memory model (–mb)	small memory model (default)
register-argument model (–mr)	stack-based model (default)
TMS320C4x model (–v40)	TMS320C3x model (default)

All code linked together in any program, including assembly language and library modules, *must* agree on these three options. Mixed model code may link without error, but it will not run correctly. The linker will issue an error message if you try to link –v40 code with –v30 code. The big and small memory models are further explained in subsection 2.3.1, page 2-30. Register-argument and stack-based models are explained in subsection 2.3.2, page 2-31. With these exceptions, compiler options including runtime model options do not affect compatibility with other modules.

Two runtime libraries are shipped with the compiler: rts30.lib and rts40.lib. Both are compiled for the small model and standard (stack-based) argument conventions. (Refer to Section 5.1 for further information.) The rts30.lib was compiled for the TMS320C3x, and rts40.lib was compiled for the TMS320C4x. If you use a model that is incompatible with these, you must build a compatible version of the runtime library with the library build utility (mk30). See Chapter 6 for more information on using this utility.

By default, the mk30 utility will create a runtime-support library called rts.lib. However, if you use more than one library, you may wish to adopt a naming convention that makes it obvious which models were used. For example, append a suffix that contains the target CPU number (30 or 40), an 'r' for the register-argument model, and a 'b' for the big model. Thus rts40rb.lib would be easily recognized as the runtime support library for the TMS320C4x, big model, with the register-argument runtime model, and rts30.lib would be for the TMS320C3x small model and stack-based or standard runtime model.

2.3.1 The Big and Small Memory Models

The small memory model (default) requires that all external variables, global variables, static variables, and compiler-generated constants in the program (after linking) fit into a single 64K word long data page (65536 words). This allows the compiler to access any of these objects without modifying the data page pointer (DP) register. Neither model restricts the size of code, automatic data, or dynamically allocated data.

The big model (–mb option) removes the 64K restriction. However, this model also forces the compiler to reload the data page pointer before accessing any external variables or compiler generated constants. This is less efficient since it forces an extra instruction and possibly one or more extra pipeline delay cycles each time the compiler accesses one of these objects.

Use the small model whenever possible. If you have large arrays, allocate them dynamically from the heap (using malloc) rather than declaring them as global or static.

Note: Size Restrictions on Small Model are not Tested

When you use the small model, you must be sure the .bss section is less than 64K words and does not crosss any 64K page boundaries. Neither the compiler nor the linker checks these restrictions against the model used.

To make sure that the .bss section conforms to these two rules, link the .bss section using the **block** attribute of the SECTIONS directive.

The statement:

```
.bss: load = block (0x10000)
```

when used with the SECTIONS directive, will force the .bss section into a 64K data page if the size of the .bss section is less than 64K. You should check the link map after linking to verify that the .bss does not violate the restrictions.

If you have variables declared in assembly language outside the .bss section, you can still use the small model, but you must access them indirectly, and you may need to use the –mf option as well. Section 4.5 on page 4-21 discusses interfacing C with assembly language.

2.3.2 The Register-Argument and Standard Models

You can choose from two different argument passing models. In the standard runtime model (default) the compiler passes all arguments to functions on the stack. When you invoke the compiler with the –mr option, some or all of the arguments are passed in registers. This results in better performance for function calls in your program. There are two important considerations when using the register-argument model:

- All functions must be prototyped. The compiler must be able to "see" the types of function parameters when the function is called. If there is a function call with no prototype visible, the compiler probably will generate incorrect code for the call. The –pf option will generate prototypes for all functions you define.
- Taking full advantage of the register-argument model requires use of the optimizer (–o option) also. The register-argument model complements the optimizer's register allocation algorithms so that function arguments can usually reside in the registers they were passed in until the function terminates.

Neither of the runtime-support libraries shipped with the compiler are compatible with the register-argument model, so you will need to build a new library to use this model. Technical details of both models are explained in Section 4.4 on page 4-16. The library build utility is described in Chapter 6.

2.4 Using the C Compiler Optimizer

The compiler package includes an optimization program that improves the execution speed and reduces the size of C programs by doing such things as simplifying loops, rearranging statements and expressions, and allocating variables into registers.

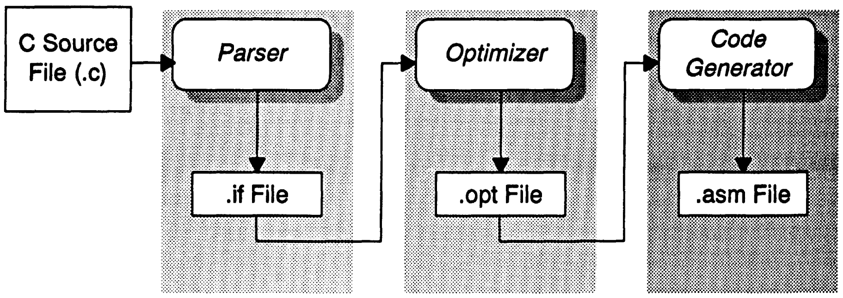
The optimizer runs as a separate pass between the parser and the code generator. The easiest way to invoke the optimizer is to use the cl30 shell program, specifying the –o option on the cl30 command line (you may also invoke the optimizer outside cl30; refer to Section 2.8 on page 2-47 for more information). The –o option may be followed by a digit specifying the level of optimization. If you do not specify a level digit, the default is level 2.

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

```
cl30 –o3 –x function.c
```

Figure 2–2 illustrates the execution flow of the compiler with standalone optimization.

Figure 2–2. Compiling a C Program with the Optimizer



The optimizer also recognizes cl30 options –s, –ma, –q, and –pk; these options are discussed in subsection 2.1.3.

To invoke the optimizer outside cl30, refer to subsection 2.8.2.

2.4.1 Optimization Levels

There are four levels of optimization: 0, 1, 2, and 3. These levels control the type and degree of optimization.

- **Level 0**
  - performs control-flow-graph simplification
  - allocates variables to registers
  - performs loop rotation
  - eliminates dead code
  - simplifies expressions and statements
  - expands calls to functions declared as inline
- **Level 1**

  performs all level 0 features, plus:
  - performs local copy/constant propagation
  - removes local dead assignments
  - eliminates local common subexpressions
- **Level 2**

  performs all level 1 features, plus:
  - performs loop optimizations
  - eliminates global common subexpressions
  - eliminates global redundant assignments
  - converts array references in loops to incremented pointer form
  - performs loop unrolling
- **Level 3**

  performs all level 2 features, plus:
  - removes all functions that are never called
  - simplifies functions that have return values that are never used
  - expands calls to small functions inline
  - reorders function definitions so 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

Note: Files That Redefine Standard Library Functions

The optimizer uses known properties of the standard library functions to perform level 3 optimizations. If you have files that redefine standard library functions, use the –ol (lowercase L) options to inform the optimizer. (See page 2-21.)

This list describes optimizations performed by the standalone optimization pass. The code generator performs several additional optimizations, particularly TMS320C3x/C4x-specific optimizations; it does so regardless of whether or not you invoke the optimizer. These optimizations are always enabled and are not affected by the optimization level you choose.

For more information about the meaning and effect of specific optimizations, refer to Appendix A.

2.4.2 Definition Controlled Inline Expansion Option (–x Option)

When the optimizer is invoked, the –x optimizer option controls inline expansion of functions that have been declared as *inline* by inhibiting or allowing the expansion of their code in place of calls. That is, code for the function will be inserted (inlined) into your function at each place it is called whenever the optimizer is invoked and the –x option is not equal to –x0. The –x2 option automatically invokes the optimizer at the default level (level 2, if the –o option is not specified separately) and defines the _INLINE preprocessor symbol as equal to 1, which causes expansion of functions declared as inline and controlled by the _INLINE symbol (For more information about _INLINE, see subsection 2.5.3, page 2-40).

Inlining makes a program faster by eliminating the overhead caused by function calls, but inlining sometimes increases code size.

For more information, see Section 2.5, *Function Inlining*, page 2-37.

2.4.3 Using the Optimizer with the Interlist Option

Optimization makes normal source interlisting impractical, because the optimizer extensively rearranges your program. Therefore, the optimizer writes reconstructed C statements (as assembly language comments), which show the optimized C statements. The comments also include a list of the allocated register variables. Note that occasionally the optimizer interlist comments may be misleading because of copy propagation or assignment of multiple or equivalent variables to the same register.

2.4.4 Debugging Optimized Code

The best way to debug code that is also optimized is to debug it in an unoptimized form and then reverify its correctness after it has been optimized.

The debugger may be used with optimized code, but the extensive rearrangement of code and the many-to-one allocation of variables to registers often makes it difficult to correlate source code with object code.

Note: Symbolic Debugging and Optimized Code

If you use the **–g** option to generate symbolic debugging information, many code generator optimizations are disabled because they disrupt the debugger. If you want to use symbolic debugging and still generate fully optimized code, use the **–mn** option on cl30; –mn re-enables the optimizations disabled by –g.

2.4.5 Special Considerations When Using the Optimizer

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

asm Statements

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 will never optimize out an asm statement (except when it is totally unreachable), the surrounding environment where the assembly code is inserted may differ significantly from its apparent context in the C source code. It is usually safe to use asm statements to manipulate hardware controls such as interrupt registers or I/O ports, but asm statements that attempt to interface with the C environment or access C variables may have unexpected results. After compilation, check the assembly output to make sure your asm statements are correct and maintain the integrity of the program.

Volatile Keyword

The optimizer analyzes data flow to avoid memory accesses whenever possible. If you have code that *depends* on memory accesses exactly as written in the C code, you *must* use the volatile keyword to identify these accesses. The compiler won't optimize out any references to volatile variables.

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

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

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

```
volatile unsigned int *ctrl
```

Aliasing

Aliasing occurs when a single object may 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 could potentially refer to any other object. The optimizer analyzes the code to determine where aliasing can and cannot occur, then optimizes as much as possible while still preserving the correctness of the program.

The compiler assumes that if the address of a local variable is passed to a function, the function might change the local by writing through the pointer but will not make its address available for use elsewhere after returning. For example, the called function cannot assign the local's address to a global variable or return it. In cases where this assumption is invalid, use the –ma option in cl30 to force the compiler to assume worst-case aliasing. In worst-case aliasing, any indirect reference may refer to such a variable.

Register Use

When you compile with the optimizer, the compiler's register usage conventions allow more variables to be stored in registers. This should not affect the correctness of normal C code, but it may invalidate some asm statements that assume that certain variables are in certain registers. Also, interrupt service routines must save all registers that may be used by C functions. The register conventions and requirements for interrupt functions are described in Section 4.3 on page 4-11.

2.5 Function Inlining

When an inline function is called and the optimizer is invoked, the code for the function is inserted at the point of the call. This 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 expansion is performed one of three ways.

- ❑ The intrinsic operators of the target system (such as abs) are inlined by the compiler by default. This happens whether or not the optimizer is used and whether or not any compiler or optimizer options are used. (You can defeat this automatic inlining by invoking the compiler with the –x0 option.)
- ❑ Definition controlled inline expansion is performed when two conditions exist:
  - ■ the *inline* keyword is encountered in source code

    and
  - ■ the optimizer is invoked (at any level)

  Functions with local static variables or a variable number of arguments will not be inlined, with the exception of functions declared as static inline. In functions defined as static inline, expansion will occur despite the presence of local statics. In addition, a limit is placed on the depth of inlining for recursive or non–leaf functions. Inlining should be used for small functions or functions that are called only a few times (though the compiler does not enforce this.)

  You can control this type of function inlining two ways:

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

  - ■ **Method 1.** By *defining* a function as *inline* within a module (with the inline keyword), you can specify that the function is inlined *within that module.* A global symbol for the function is created, but the function will be inlined only within the module where it is defined as inline. It will be called by other modules unless they contain a compatible static inline declaration. Functions defined as inline are expanded when the optimizer is invoked and the –x option is not equal to –x0. Setting the –x option to –x2 will automatically invoke the optimizer at the default level (level 2).

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

- **Method 2.** By *declaring* a function as static inline, you can specify that the function is inlined in the present module. This names the function and specifies that the function is to be expanded inline, but no code is generated for the function declaration itself. Functions declared in this way may be placed in header files and included by all source modules of the program. Declaring a function as static inline in a header file specifies that the function is inlined in any module that includes the header.

  Functions declared as inline are expanded whenever the optimizer is invoked at any level. Functions declared as inline and controlled by the _INLINE preprocessor symbol, such as the runtime library functions, are expanded whenever the optimizer is invoked and the _INLINE preprocessor symbol is equal to 1. When you define an inline function, it is recommended that you use the _INLINE preprocessor symbol to control its declaration. If you fail to control the expansion using _INLINE, and subsequently compile *without* the optimizer, the call to the function will be unresolved. For more information, see subsection 2.5.3, *The _INLINE Preprocessor Symbol,* page 2-40.

- Automatic inline expansion (functions *not* declared as inline) is done when the optimizer is invoked at level 3. By default, the optimizer will inline very small functions. You can change the size of functions that are automatically inlined with the **–oi***size* option. The –oi option specifies that functions whose size (times number of calls) is less than *size* units are inlined regardless of how they were declared. The optimizer measures the size of a function in arbitrary units. However, the size of each function is reported in the optimizer information file (–on1 option). If you want to be certain that a function is always inlined, use the inline keyword (discussed above and in the next subsection). You can defeat all automatic inlining of small functions not declared as inline by setting the size to 0 (–oi0).

Note: Function Inlining May Greatly Increase Code Size

It should be noted that expanding functions inline expands code size, and that inlining a function that is called a great number of times can expand code size exponentially. Function inlining is optimal for functions that are called only a small number of times, or for small functions that are called more often. If your code size seems too large, try compiling with the –x0 keyword and note the difference in code size.

2.5.1 Controlling Inline Expansion (–x Option)

A command line switch controls the types of inline expansion performed.

–x0 : no inline expansion. Defeats the default expansions listed below.

–x1 : is the default value. The intrinsic operators are inlined wherever they are called. This is true whether or not the optimizer is invoked, and whether or not a –x option is specified (except –x0). Intrinsic operators are:

- abs
- labs
- fabs

–x2/–x: creates the preprocessor symbol _INLINE, assigns it the value 1, and invokes the optimizer at level 2, thereby enabling definition controlled inline expansion.

If a function has been defined or declared as inline, it will be expanded inline whenever the optimizer is called and the –x option is not equal to –x0. Setting the –x option to –x2 automatically invokes the optimizer and thus causes the automatic expansion of functions defined or declared as inline, as well as causing the other optimizations defined at level 2, which is the default level for the optimizer. The _INLINE preprocessor symbol has been used to control the expansion of the runtime library modules. They will be expanded if _INLINE is equal to 1, but will be called if _INLINE is not equal to 1. Other functions may be set up to use _INLINE in the same way. For more information, see subsection 2.5.3, *The _INLINE Preprocessor Symbol.*

If the –x, –x1 or –x2 option is used together with the –o option at any level, the optimizer will be invoked at the level specified by –o rather than at the level specified by –x.

2.5.2 Automatic Inline Expansion Option (–oi*size* Option)

The optimizer will automatically inline all small functions (not defined or declared with the *inline* keyword) when invoked at level 3. A command line option controls the size of functions inlined when the optimizer is invoked at level 3. The –oi option can be used three ways:

- ❑ If you set the size parameter to zero (–oi0), all size controlled inlining is disabled.
- ❑ If you do not use the –oi option, the optimizer inlines very small functions.
- ❑ If you set the size parameter to a nonzero integer, the optimizer will inline all functions whose size is less than the *size* parameter. If the function is

called more than once, the optimizer multiplies the size of the function by the number of calls, and will inline the function only if the resulting product is less than the *size* parameter. The optimizer measures the size of a function in arbitrary units. The optimizer information file (created with the –on1 or –on2 option), however, will report the size of each function in the same units that the –oi option uses.

2.5.3 _INLINE Preprocessor Symbol

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

The _INLINE symbol is used in the string.h header file to declare the function correctly, regardless of whether inlining is used. The _INLINE symbol is turned off in the memcpy source *before* the header file is included, because it is unknown whether the rest of the module is compiled with inlining.

If the rest of the modules are compiled with inlining enabled *and* the string.h header is included, all references to *memcpy* will be inlined and the linker will not have to use the *memcpy* in the runtime-support library to resolve any references. Otherwise, the runtime-support library code will be used to resolve the references to *memcpy* and function calls will be generated.

You will want to use the _INLINE preprocessor symbol in the same way so that your programs will run regardless of whether inlining mode is selected for any or all of the modules in your program.

Example 2–1 on page 2-41 illustrates how the runtime support library uses the _INLINE symbol.

Example 2–1. How the Runtime Support Library Uses the _INLINE Symbol

```
/****************************************************************************/
/* STRING.H HEADER FILE                                                     */
/****************************************************************************/
typdef unsigned size_t

#if _INLINE
#define __INLINE static inline    /* Declaration when inlining      */
#else
#define __INLINE                  /*No declaration when not inlining*/
#endif

__INLINE  void *memcpy (void *_s1, const void *_s2, size_t _n);

#if _INLINE                       /* Declare the inlined function   */

static inline void *memcpy (void *to, const void *from, size_t n)
{
   register char *rto  = (char *) to;
   register char *rfrom   = (char *) from;
   register size_t rn;

   for (rn = 0; rn < n; rn++) *rto++ =rfrom++;
   return (to);
}

#endif    /* _INLINE      */

#undef __INLINE
```

```
/****************************************************************************/
/* MEMCPY.C      (rtsxx.lib)                                                */
/****************************************************************************/
#undef _INLINE      /* Turn off so code will be generated          */

#include <string.h>

void *memcpy (void *to, const void *from, size_t n)
{
   register char *rto  = (char *) to;
   register char *rfrom   = (char *) from;
   register size_t rn;

   for (rn = 0; rn < n; rn++) *rto++ =rfrom++;
   return (to);
}
```

There are two definitions of the memcpy function. The first, in the header file is an inline definition. Note how this definition is enabled and the prototype declared as static inline only if _INLINE is true: that is, the module including this header is compiled with the –x option.

The second definition (in memcpy.c) is for the library so that the callable version of memcpy exists when inlining is disabled. Since this is not an inline function, the _INLINE symbol is undefined (#undef) before including string.h so that the non-inline version of memcpy's prototype is generated.

If the application is compiled with the –x option *and* the string.h header is included, all references to memcpy in the runtime support library will be inlined and the linker will not have to use the memcpy in the runtime support library to resolve any references. Any modules that call memcpy that are not compiled with inlining enabled will generate calls that the linker resolves by getting the memcpy code out if the library.

You will want to use the _INLINE preprocessor symbol in the same way so that your programs will run regardless of whether inlining mode is selected for any or all of the modules in your program.

2.6 Using the Interlist Utility

The compiler package includes a utility that interlists your original C source statements into the assembly language output of the compiler. The interlist utility enables you to inspect the assembly code generated for each C statement.

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

```
cl30 –s function
```

The interlist utility runs as a separate pass between the code generator and the assembler. It reads both the assembly and C source files, merges them, and writes the C statements into the assembly file as comments beginning with ;>>>>. The output assembly file, function.asm, is assembled normally. The –s option automatically prevents the cl30 shell from deleting the interlisted assembly language file (as if you had used –k).

Example 2–2 shows a typical interlisted assembly file.

Example 2–2. An Interlisted Assembly Language File

```
;>>>>           main()
;>>>>               int i, j;
********************************************
*        FUNCTION DEF : _main              *
********************************************
_main:
        PUSH   FP
        LDI    SP,FP
        ADDI   2,SP
;>>>>          i += j;
        LDI    *+FP(2),R0
        ADDI   R0,*+FP(1),R1
        STI    R1, *+FP(1)
;>>>>          j = i + 123;
        ADDI   123,R1
        STI    R1,*+FP(2)
```

To invoke the interlist utility outside of the shell, refer to subsection 2.8.4.

Note: Using the –s Option With the Optimizer

Optimization makes normal source interlisting impractical because the optimizer extensively rearranges your program. Therefore, when you use the –s option, the optimizer writes reconstructed C statements. The comments also include a list of the allocated register variables. Note that occasionally the optimizer interlist comments may be misleading because of copy propagation or assignment of multiple or equivalent variables to the same register.

2.7 How the Compiler Handles Errors

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

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

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

Errors are divided into 4 classes according to severity; these classes are identified by the letters *W, E, F,* and *I (upper-case i).*

Code-W Error Messages

Code-W errors are warnings. They result from a condition that is technically undefined according to the rules of the language, and code may not generate what you intended. This is an example of a code-W error:

```
"file.c", line 42: [W063] illegal type for register variable 'x'
```

Code-E Error Messages

Code-E errors are recoverable. They result from a condition that violates the semantic rules of the language. Although these are normally fatal errors, the compiler can recover and generate an output file if you use the –pe option. Refer to subsection 2.7.1 for more information. This is an example of a code-E error:

```
"file.c", line 66: [E056] illegal storage class for function 'f'
```

Code-F Error Messages

Code-F errors are fatal. They result from a condition that violates the syntactic or semantic rules of the language. The compiler cannot recover and therefore does not generate output for code-F errors. This is an example of a code-F error:

```
"file.c", line 71: [F090] structure member 'a' undefined
```

Code-I Error Messages

Code-I errors are implementation errors. They occur when one of the compiler's internal limits is exceeded. These errors are usually caused by extreme behavior in the source code rather than by explicit errors. In most cases, code-I errors cause the compiler to abort immediately. Most code-I messages contain the maximum value for the limit that was exceeded. (Those limits that are absolute are also listed in Section 3.8 on page 3-12.) This is an example of a code-I error:

```
"file.c", line 99: [I015] block nesting too deep (max=20)
```

Other Error Messages

The compiler also reports other errors, such as incorrect command line syntax or inability to find specified files.These errors are usually fatal and are identified by the symbol >> preceding the message.

This is an example of such an error:

```
>> Cannot open source file 'mystery.c'
```

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

A *fatal error* is an error that prevents the compiler from generating an output file. Normally, code-E, -F, and -I errors are fatal, while -W errors are not fatal. The –pe shell option causes the compiler to effectively treat code-E errors as warnings, so that the compiler will generate code for the file despite the error.

Using –pe allows you to bend the rules of the language, so be careful. As with any warning, the compiler may not generate what you expect.

Note that there is no way to specify recovery from code-F or -I errors; these are always fatal and prevent generation of a compiled output file.

2.7.2 Suppressing Warning Messages (–pw Option)

The –pw option enables you to suppress the output of warning messages, causing the compiler to quietly ignore code-W messages (warnings). This is useful when you are aware of the condition causing the warning and consider it innocuous.

You can double –pw options to suppress code-E errors as well: doubling the –pw option suppresses both code-W and code-E errors. Doubling –pw is useful in conjunction with the –pe option so that code-E errors are ignored completely. For example:

```
cl30 –peww *.c  ; completely ignore all W and E errors
```

2.7.3 An Example of How You Can Use Error Options

The following example demonstrates how the –pe and –pw options can be used to suppress errors and error messages. The examples use this 2-line code segment:

```
int *pi; char *pc;
pi = pc;
```

- ❑ If you invoke the code with the shell (and –q), this is the result:

  ```
  [err]
  "err.c", line3:

          [E104] operands of '=' point to different types
  ```

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

- ❑ If you invoke the code with the shell and the –pe option, this is the result:

  ```
  [err]
  "err.c", line3:

          [E104] operands of '=' point to different types
  ```

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

- ❑ If you invoke the code with the shell and –peww (–pe and double –pw), this is the result:

  ```
  [err]
  ```

  As in the previous case, –pe causes the compiler to overlook the error and generate code. Because the two –pw options are used, the message is suppressed.

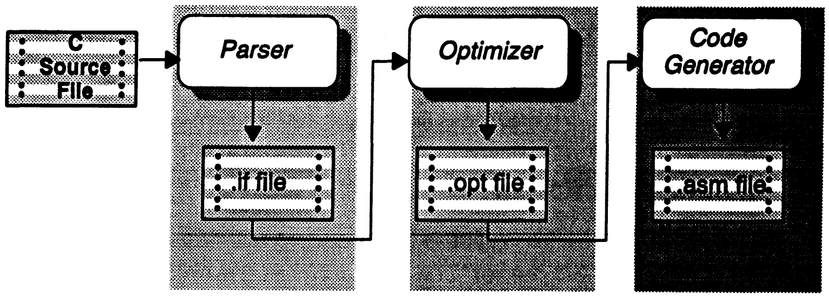
2.8 Invoking the Tools Individually

The TMS320 floating-point C compiler offers you the versatility of invoking all of the tools at once, using the shell, or invoking each of the tools individually. To satisfy a variety of applications, you can invoke the compiler (parser, optional optimizer, and code generator), the assembler, and the linker as individual programs. This section also describes how to invoke the interlist utility outside the shell.

Compiler

The compiler is made up of three distinct programs: the parser, optimizer, and code generator.

Figure 2–3. Compiler Overview



The input for the **parser** is a C source file. The parser reads the source file, checking for syntax and semantic errors, and writes out an internal representation of the program called an intermediate file. Subsection 2.8.1, page 2-48, describes how to run the parser, and also describes how to run the parser in two passes: the first pass preprocesses the code, and the second pass parses the code.

The **optimizer** is an optional pass that runs between the parser and the code generator. The input is the intermediate file (.if) produced by the parser. When you run the optimizer, you choose the level of optimization. The optimizer performs the optimizations on the intermediate file and produces a highly efficient version of the file in the same intermediate file format. Section 2.4, page 2-32, describes the optimizer.

The input for the **code generator** is the intermediate file produced by the parser (.if) or the .opt file from the optimizer. The code generator produces an assembly language source file. Subsection 2.8.3, page 2-51, describes how to run the code generator.

Assembler

The input for the **assembler** is the assembly language file produced by the code generator. The assembler produces a COFF object file. The assembler is described fully in the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide.*

Interlist Utility

The inputs for the interlist utility are the assembly file produced by the compiler and the C source file. The utility produces an expanded assembly source file containing statements from the C file as assembly language comments. Section 2.6 on page 2-43 describes the interlisting file and subsection 2.8.4 on page 2-53 describes the use of the interlist utility.

Linker

The input for the **linker** is the COFF object file produced by the assembler. The linker produces an executable object file. Section 2.9 describes how to run the linker. The linker is described fully in the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide.*

2.8.1 Invoking the Parser

The first step in compiling a TMS320 floating-point C program is to invoke the C parser. The parser reads the source file, performs preprocessing functions, checks the syntax, and produces an intermediate file that can be used as input for the code generator.

To invoke the parser, enter:

```
ac30 input file [output file] [options]
```

ac30	is the command that invokes the parser.
input file	names the C source file that the parser uses as input. If you don't supply an extension, the parser assumes that the file's extension is *.c.*
output file	names the intermediate file that the parser creates. If you don't supply a filename for the output file, the parser uses the input filename with an extension of *.if.*
options	affect parser operation. Each option available for the standalone parser has a corresponding shell option that performs the same function. Table 2–3 shows the shell options, the parser options, and the corresponding functions.

Table 2–3. Parser Options and Shell Options

ac30 Option	Shell Option	Function
–?	–p?	enable trigraph expansion
–d*name* [=def]	–d*name* [=def]	predefine macro name
–e	–pe	treat code-E errors as warnings
–f	–pf	prototype declared functions
–i *dir*	–i *dir*	define #include search path
–k	–pk	allow K&R compatibility
–l (lowercase L)	–pl	generate .pp file
–mr	–mr	use register-argument runtime model
–mb	–mb	use big memory model
–n	–pn	suppress #line directives
–o	–po	preprocess only
–q	–q	suppress progress messages (quiet)
–tf	–tf	relax prototype checking
–tp	–tp	relax pointer combination
–u*name*	–u*name*	undefine macro name
–v*xx*	–v*xx*	select CPU version
–w	–pw	suppress warning messages
–x0	–x0	disable inline expansion
	–x1	expand *abs, fabs,* and *labs* inline (default)
–x	–x or –x2	#define _INLINE

Parsing in Two Passes

Compiling very large source programs on small host systems such as PCs can cause the compiler to run out of memory and fail. You may be able to work around such host memory limitations by running the parser as two separate passes: the first pass preprocesses the file, and the second pass parses the file.

When you run the parser as one pass, it uses host memory to store both macro definitions and symbol definitions simultaneously. But when you run the parser as two passes, these functions can be separated. The first pass performs only preprocessing, therefore memory is needed only for macro definitions. In the second pass, there are no macro definitions, therefore memory is needed only for the symbol table.

The following example illustrates how to run the parser as two passes:

1) Run the parser with the –po option, specifying preprocessing only.

   ```
   cl30 -po file.c
   ```

   If you want to use the –d, –mr, –u, –v, –x, or –i options, use them on this first pass. This pass produces a preprocessed output file called file.pp. For more information about the preprocessor, refer to Section 2.2.

2) Rerun the whole compiler on the preprocessed file to finish compiling it.

   ```
   cl30 file.pp
   ```

   You can use any other options on this final pass.

2.8.2 Optimizing Parser Output

The second step in compiling a TMS320 floating-point C program — optimizing— is optional. After parsing a C source file, you can choose to process the intermediate file with the optimizer. The optimizer improves the execution speed and reduces the size of C programs. The optimizer reads the intermediate file, optimizes it according to the level you choose, and produces an intermediate file. The intermediate file has the same format as the original intermediate file, but it enables the code generator to produce more efficient code.

To invoke the optimizer, enter:

opt30 [*input file* [*output file*]] [*options*]

opt30	is the command that invokes the optimizer.
input file	names the intermediate file produced by the parser. The optimizer assumes that the extension is *.if.*
output file	names the intermediate file that the optimizer creates. If you don't supply a filename for the output file, the optimizer uses the input filename with an extension of *.opt.*
options	affect the way the optimizer processes the input file. The options that you use in standalone optimization are the same as those used for the shell. Besides the optimizer options, some other compiler options may be used with the optimizer. Those formats are shown in Table 2–4. Section 2.4 provides a detailed list of the optimizations performed at each level. Detailed explanations of each optimizer option may be found on page 2-21.

Table 2–4. Optimizer Options and cl30 Options

opt30 Option	Shell Option	Function
–a	–ma	assume variables are aliased
–f	–mf	force indirect access to externals
–k	–pk	allow K&R compatibility
–m	–mm	enable short multiply ('C3x only)
–o0	–o0	level 0 register optimization
–o1	–o1	level 1 plus local optimization
–o2	–o2	level 2 plus global optimization
–q	–q	suppress progress messages (quiet)
–r	–mr	use register-argument runtime model
–s	–s	interlist C source
–v*xx*	–v*xx*	select CPU version
–x	–mx	support first pass TMX silicon
–z		retains input file†

† –z tells the optimizer to retain the input file (the intermediate file created by the parser). Note that if you do not specify the –z option, the optimizer deletes the input file.

2.8.3 Invoking the Code Generator

The third step in compiling a TMS320 floating-point C program is to invoke the C code generator. As Figure 2–3 on page 2-47 shows, the code generator converts the intermediate file produced by the parser into an assembly language source file for input to the TMS320 floating-point assembler. You can modify this output file or use it as input for the assembler. The code generator produces re-entrant relocatable code, which, after assembling and linking, can be stored in ROM.

To invoke the code generator as a standalone program, enter:

```
cg30 [input file [output file [tempfile]]] [options]
```

cg30	is the command that invokes the code generator.
input file	names the intermediate file that the code generator uses as input. If you don't supply an extension, the code generator assumes that the extension is *.if*. If you don't specify an input file, the code generator will prompt you for one.

output file	names the assembly language source file that the code generator creates. If you don't supply a filename for the output file, the code generator uses the input filename with the extension of *.asm.*
tempfile	names a temporary file that the code generator creates and uses. The default filename for the temporary file is the input filename appended to an extension of *.tmp.* The code generator deletes this file after using it.
options	affect the way the code generator processes the input file. Each option available for the standalone code generator mode has a corresponding shell option that performs the same function. The following table shows the shell options, the code generator options, and the corresponding functions.

Table 2–5. Code Generator Options and Shell Options

cg30 Option	Shell Option	Function
–a	–ma	assume aliased variables
–b	–mb	enable the big memory model
–i	–mi	disables RPTS instructions for loops (uses RPTB)
–m	–mm	enable short multiply ('C3x only)
–n	–mn	re-enable optimizations disabled by symbolic debugging
–o	–g	enable symbolic debugging
–p	–mr	use register-argument runtime model
–q	–q	suppress progress messages (quiet)
–x	–mx	avoid TMX silicon bugs
–z		retain the input file

† **–z** tells the code generator to retain the input file (the intermediate file created by the parser). This option is useful for creating several output files with different options; for example, you might want to use the same intermediate file to create one file that contains symbolic debugging directives (–o option) and one that doesn't. Note that if you do not specify the –z option, the code generator deletes the input (intermediate) file.

2.8.4 Invoking the Interlist Utility

Note: Interlisting With the Shell Program and the Optimizer

You can create an interlisted file by invoking the shell program with the –s option. Anytime that you request interlisting on optimized code, the optimizer, not the interlist utility, performs the interlist function.

The fourth step in compiling a TMS320 floating-point C program is optional. After you have compiled a program, you can run the interlist utility. To run the interlist utility from the command line, the syntax is:

clist *asmfile* [*outfile*] [*–options*]

clist	is the command that invokes the interlist utility.
asmfile	is the filename of assembly language output from the compiler.
outfile	names the interlisted output file. If you omit this, the file has the same name as the assembly file with an extension *.cl*.
options	control the operation of the utility as follows:

–b	removes blanks and useless lines (lines containing comments or lines containing only *{* or *}*).
–r	removes symbolic debugging directives.
–q	removes banner and status information.

The interlist utility uses .line directives, produced by the code generator, to associate assembly code with C source. For this reason, when you compile the program, you *must* use the –g shell option to specify symbolic debugging if you want to interlist it. If you do not want the debugging directives in the output, use the –r interlist option to remove them from the interlisted file.

The following example shows how to compile and interlist function.c. To compile, enter:

```
cl30 –gk –mn function
```

This compiles and produces symbolic debugging directives, keeps the assembly language file, and allows normal optimization.

To produce an interlist file, enter

```
clist –r function
```

This interlists and removes debugging directives from the file. The output from this example is function.cl.

2.9 Linking C Code

The TMS320 floating-point C compiler and assembly language tools provide two methods for linking your programs:

- you can link object files using the linker alone.
- you can compile, assemble, and link in one step by using the shell. This is useful when you have a single source module.

This section describes how to invoke the linker with each method. It also discusses special requirements of linking C code: including the runtime-support libraries, specifying the initialization model, and allocating the program into memory.

2.9.1 Invoking the Linker

The TMS320 floating-point C compiler and assembly language tools support modular programming by allowing you to compile and assemble individual modules and then link them together.

The general syntax for invoking the linker is:

lnk30 [*–options*] *filename*$_1$ ... *filename*$_n$

lnk30	is the command that invokes the linker.
options	can appear anywhere on the command line or in a linker command file. (Options are discussed in Table 2–1, page 2-6.)
filenames	can be object files, linker command files, or archive libraries. The default extension for all input files is *.obj*; any other extension must be explicitly specified. The linker can determine whether the input file is an object file or an ASCII file that contains linker commands. The default output filename is *a.out.*

Linker options can be specified on the command line or in a command file. The command file allows you to use the MEMORY and SECTIONS directives, which can customize your memory to your specifications. If you enter the lnk30 command with no options, the linker will prompt for them.

```
Command files :
Object files [.obj] :
Output files [ ] :
Options :
```

This is the usual syntax for linking compiler-based C programs as a separate step:

```
lnk30 –c filenames    –o name.out    –l rts.lib
                  or
lnk30 –cr filenames   –o name.out    –l rts.lib
```

lnk30	is the command that invokes the linker.
–c/–cr	are options that tell the linker to use special conventions defined by the C environment. Note that when you use the shell to link, it automatically passes –c to the linker.
filenames	are object files created by compiling and assembling C programs.
–o *name.out*	names the output file. If you don't use the –o option, the linker creates an output file with the default name of ***a.out***.
–l *rts.lib*	identifies the appropriate archive library containing C runtime-support and floating-point math functions. (The –l option tells the linker that a file is an object library.) If you're linking C code, you must use a runtime-support library. The rts.lib library is included with the C compiler.

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

```
lnk30 –c prog1 prog2 prog3 –l rts30.lib –o prog.out
```

The linker uses a default allocation algorithm to allocate your program into memory. You can use the MEMORY and SECTIONS linker directives to customize the allocation process, and you can also use other linker options, described in Table 2–1 on page 2-6. These directives are also described in Chapter 8 of the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide.*

2.9.2 Using the Shell to Invoke the Linker (–z Option)

The options and parameters discussed in this section apply to both methods of linking; however,when you are linking with the shell, the options follow the –z shell option.

By default, the shell does not run the linker. However, if you use the –z option, the shell compiles, assembles, and links in one step. When using –z to enable linking, remember that:

- ❑ –z must follow all source files and compiler options on the command line (or be specified with C_OPTION)
- ❑ –z divides the command line into compiler options (before –z) and linker options (following –z)
- ❑ –c and –n suppress –z, so do not use them if you want to link

All arguments that follow –z on the command line are passed to the linker. These arguments can be linker options (listed in Table 2–1 on page 2-6), linker command files, additional object files, linker options, or libraries.

For example, to compile and link all the .c files in a directory, enter:

```
cl30 –sq *.c –z c.cmd –o prog.out –l rts30.lib
```

The shell compiles all of the files in the current directory that have a .c extension under the –sq option. When the compiler is finished, it invokes the linker (–z option.) The linker, using options specified in the c.cmd command file, will link together all of the object files produced by the compiler run, the runtime support library, and any additional files specified in c.cmd. The resulting linked file will be called prog.out.

The order in which the linker processes arguments can be important, especially for command files and libraries. The cl30 shell passes arguments to the linker in the following order:

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

–c and –n Shell Options

You can override the –z option by using the –c or –n shell options. The –c option is especially helpful when you have specified –z in the C_OPTION environment variable and want to selectively disable linking with –c on the command line. The –n option causes the shell to stop the process after compilation.

The –c linker option has a different function than, and is independent of, the –c shell option. By default, the shell automatically uses the –c linker option that tells the linker to use C linking conventions (ROM model of initialization). If you want to use –cr (RAM model of initialization) rather than –c, you can pass –cr as a linker option instead.

2.9.3 Controlling the Linking Process

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

- ❑ include the compiler's runtime-support library
- ❑ specify the initialization model
- ❑ determine how you want to allocate your program into memory

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

For more information about how to operate the linker, refer to Chapter 8 of the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide.*

Runtime-Support Libraries

All C programs must be linked with a runtime-support library. This archive library contains standard C library functions (such as malloc and strcpy) as well as functions used by the compiler to manage the C environment. To link a library, simply use the –l option on the command line:

```
lnk30 –c  filenames   –l  rts30.lib –l  flib30.lib ...    or
lnk30 –cr  filenames  –l  rts30.lib –l  flib30.lib ...
```

Two versions of the standard C runtime-support libraries are included with the compiler: rts30.lib for TMS320C3x programs and rts40.lib for TMS320C4x programs.

Generally, the libraries should be specified as the last filename 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 will not be resolved. You can use the –x option to force the linker to reread all libraries until references are resolved. Whenever you specify a library as linker input, the linker includes and links only those library member that resolve undefined references.

All C programs must be linked with an object module called *boot.obj.* When a program begins running, it executes boot.obj first. boot.obj contains code and data for initializing the runtime environment; the linker automatically extracts boot.obj and links it when you use –c or –cr and include rts.lib in the link.

Chapter 5 describes additional runtime-support functions that are included in rts.lib. These functions include ANSI C standard runtime support.

Initialization (RAM and ROM Models)

The C compiler produces tables of data for autoinitializing global variables. The format of these tables is discussed on page 4-34. These tables are in a named section called *.cinit*. The initialization tables can be used in either of two ways:

❑ **RAM Model** (–cr linker option)

  Global variables are initialized at *load time*. Use the –cr linker option to select the RAM model. For more information about the RAM model, refer to page 4-35.

❑ **ROM Model** (–c linker option)

  Global variables are initialized at *runtime*. Use the –c linker option to select the ROM model. For more information about the ROM model, refer to page 4-36.

The –c and –cr Linker Options

Whenever you link a C program, you must use either the –c or the –cr option. These options tell the linker to use special conventions required by the C environment; for example, they tell the linker to select the ROM or RAM model of autoinitialization. Note that when you use the shell to link programs, the –c option is the default. 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 runtime-support library rts30.lib or rts40.lib.

❑ The **.cinit** output section is padded with a termination record so that the loader (RAM model) or the boot routine (ROM model) knows when to stop reading the initialization tables.

❑ In the **RAM model** (–cr option),

  ■ The linker sets the symbol cinit to –1. This indicates that the initialization tables are not in memory, so no initialization is performed at runtime.

  ■ The STYP_COPY flag (010h) is set in the .cinit section header. STYP_COPY is the special attribute that tells the loader to perform autoinitialization directly and not to load the .cinit section into memory. The linker does not allocate space in memory for the .cinit section.

Note that a loader is not included as part of the C compiler package.

- ❑ In the **ROM model** (–c option), the linker defines the symbol cinit as the starting address of the .cinit section. The C boot routine uses this symbol as the starting point for autoinitialization.

Sections Created by the Compiler

The compiler produces five 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 2–6 summarizes the sections.

Table 2–6. Sections Created by the Compiler

Name	Type	Contents
.bss	Uninitialized	global and static variables
.cinit	Initialized	initialization values for explicitly initialized global and static variables
.const	Initialized	global and static constant variables that are explicitly initialized and string literals
.stack	Uninitialized	software stack
.text	Initialized	executable code and floating-point constant
.sysmem	Uninitialized	memory for malloc functions

When you link your program, you must specify where to locate the sections in memory. In general, initialized sections can be linked into ROM or RAM; uninitialized sections must be linked into RAM. Refer to subsection 4.1.1 on page 4-2 for a complete description of how the compiler uses these sections. The linker provides MEMORY and SECTIONS directives for performing this process.

For more information about allocating sections into memory, refer to Chapter 9, in the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide.*

Sizing the Stack and Heap

The linker provides two options that allow you to specify the size of the .stack and .sysmem sections.

–stack*xxxx* sets the size of the .stack section to *xxxx* words. The value *xxxx* must be constant.

–heap*xxxx* sets the size of the .sysmem section to *xxxx* words. The value *xxxx* must be constant.

The linker always includes the stack section; and includes the .sysmem section only if you use memory allocation functions (such as malloc()). The linker resizes these sections only if the value specified by the option is larger than the input section size (in the standard library, the size is zero, so any –stack or –heap option takes effect). The default size for both sections is 1K (1024) words.

Allocating the .bss and .const Section

When you use the small memory (default) model, the sum of the *entire* .bss and .const section must fit within a single 64K word long data page *and* must not cross any 64K address boundaries. Use the linker's **block** qualifier to help guarantee the correct allocation of .bss and .const.

Sample Linker Command File

The compiler package includes two sample linker command files called c30.cmd and c40.cmd that can be used to link C programs. To link your program using a command file, enter the following command:

```
lnk30 object.files –o output.file –m map.file c30.cmd
```

Figure 2–4 shows the contents of c30.cmd. Figure 2–5 shows the contents of c40.cmd.

To link your program using the TMS320C4x command file, enter the following command:

```
lnk30 object.files –o output.file –m map.file c40.cmd
```

Figure 2–4. Sample Linker Command File for TMS320C3x C Programs

```
/****************************************************************************/
/* C30.CMD -  v4.60   COMMAND FILE FOR LINKING C30 C PROGRAMS               */
/*                                                                          */
/*    Usage:        lnk30 <obj files...> –o <out file> –m <map file> c30.cmd */
/*                                                                          */
/*    Description: This file is a sample command file that can be used     */
/*                 for linking programs built with the TMS320C30 C          */
/*                 Compiler.  Use it 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)   Be sure to use the right library! Use a library that     */
/*                 matches the runtime model you are using.                 */
/*                                                                          */
/*           (2)   You must specify the directory in which rts.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)   When using the small (default) memory model, be sure     */
/*                 that the ENTIRE .bss section fits within a single page.  */
/*                 To satisfy this, .bss must be smaller than 64K words and */
/*                 must not cross any 64K boundaries.                       */
/****************************************************************************/
–c                                              /* LINK USING C CONVENTIONS     */
–stack 0x400                                     /* 1K STACK                    */
–heap  0x400                                     /* 1K HEAP                     */
–lrts30.lib                                      /* GET RUN–TIME SUPPORT        */

 /* SPECIFY THE SYSTEM MEMORY MAP */

 MEMORY
 {
    ROM:   org = 0x0       len = 0x1000     /* INTERNAL 4K ROM                 */
    EXT0:  org = 0x1000    len = 0x7ff000   /* EXTERNAL MEMORY                 */
    XBUS:  org = 0x800000 len = 0x2000     /* EXPANSION BUS                   */
    IOBUS: org = 0x804000 len = 0x2000     /* I/O BUS                         */
    RAM0:  org = 0x809800 len = 0x400      /* RAM BLOCK 0                     */
    RAM1:  org = 0x809c00 len = 0x400      /* RAM BLOCK 1                     */
    EXT1:  org = 0x80a000 len = 0x7f6000   /* EXTERNAL MEMORY                 */
 }

 /* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

 SECTIONS
 {
    .text:    > EXT0                  /* CODE                                  */
    .cinit:   > EXT0                  /* INITIALIZATION TABLES                 */
    .const:   > EXT0                  /* CONSTANTS                             */
    .stack:   > RAM0                  /* SYSTEM STACK                          */
    .sysmem:  > RAM1                  /* DYNAMIC MEMORY (HEAP)                 */
    .bss:     > EXT1, block 0x10000   /* VARIABLES                             */
 }
```

Figure 2–5. Sample Linker Command File for TMS320C4x C Programs

```
/****************************************************************************/
/* C40.CMD -  v4.60   COMMAND FILE FOR LINKING C40 C PROGRAMS             */
/*                                                                        */
/*    Usage:        lnk30 <obj files...> –o <out file> –m <map file> c40.cmd */
/*                                                                        */
/*    Description: This file is a sample command file that can be used    */
/*                 for linking programs built with the TMS320C40 C        */
/*                 Compiler.  Use it 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)   Be sure to use the right library!  Use a library that  */
/*                 matches the runtime model you are using.               */
/*                                                                        */
/*           (2)   You must specify the directory in which rts.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)   When using the small (default) memory model, be sure   */
/*                 that the ENTIRE .bss section fits within a single page. */
/*                 To satisfy this, .bss must be smaller than 64K words and */
/*                 must not cross any 64K boundaries.                     */
/****************************************************************************/
–c                                         /* LINK USING C CONVENTIONS      */
–stack 0x400                               /* 1K STACK                      */
–heap  0x400                               /* 1K HEAP                       */
–lrts40.lib                                /* GET RUN–TIME SUPPORT          */

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
   ROM:    org = 0x000000  len = 0x1000     /* INTERNAL ROM                 */
   RAM0:   org = 0x2FF800  len = 0x400      /* RAM BLOCK 0                  */
   RAM1:   org = 0x2FFC00  len = 0x400      /* RAM BLOCK 1                  */
   LOCAL:  org = 0x300000  len = 0x7D00000  /* LOCAL BUS                    */
   GLOBAL: org = 0x8000000 len = 0x8000000  /* GLOBAL BUS                   */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
   .text:    > LOCAL                  /* CODE                                */
   .cinit:   > LOCAL                  /* INITIALIZATION TABLES               */
   .const:   > LOCAL                  /* CONSTANTS                           */
   .stack:   > RAM0                   /* SYSTEM STACK                        */
   .sysmem:  > RAM1                   /* DYNAMIC MEMORY (HEAP)               */
   .bss:     > LOCAL, block 0x10000   /* VARIABLES                           */
}
```

In these command files, the –c option specifies that the linker use the ROM initialization model, and the –l (lowercase "L") option with rts30.lib (or rts40.lib) specifies that the linker search for the library named rts30.lib (or rts40.lib). If the library is not in the current directory, you can customize the command file to use C_DIR or the –i option to define a path where the rts40.lib can be found.

You will most likely have to customize the command file to fit a particular application by adding or modifying options, libraries, memory configurations, and section allocations. If you use the RAM initialization model, change –c to –cr. If you use the big memory model, you can remove the 64K-word blocking on the .bss section.

For more information about operating the linker, refer to the linker chapter in the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide.*

Chapter 3

TMS320C3x/C4x C Language

The TMS320 floating-point C compiler supports the C language based on the ANSI (American National Standards Institute) C standard. This standard was developed by a committee chartered by ANSI to standardize the C programming language.

ANSI C supersedes the de facto C standard, which was described in the first edition of *The C Programming Language* and based on the UNIX System V C language. The ANSI standard is described in the American National Standard for Information Systems–Programming Language C X3.159-1989. The second edition of *The C Programming Language*, by Kernighan and Ritchie, is based on the ANSI standard and is used here as a reference. ANSI C encompasses many of the language extensions provided by recent C compilers and formalizes many previously unspecified characteristics of the language.

The TMS320 floating-point C compiler strictly conforms to the ANSI C standard. The ANSI standard identifies certain implementation-defined features that may differ from compiler to compiler, depending on the type of processor, the runtime environment, and the host environment. This chapter describes how these and other features are implemented for the TMS320 floating-point C compiler.

These are the topics covered in this chapter:

Topic **Page**

3.1 Characteristics of TMS320 Floating-Point C

The ANSI standard identifies some features of the C language that are affected by characteristics of the target processor, runtime environment, or host environment. For reasons of efficiency or practicality, this set of features may differ among standard compilers. This section describes how these features are implemented for the TMS320 floating-point C compiler.

The following list identifies all such cases and describes the behavior of the TMS320 floating-point C compiler in each case. Each description also includes a reference to the formal ANSI standard and to the *The C Programming Language* by Kernighan and Ritchie (K&R).

Identifiers and constants

- The first 31 characters of all identifiers are significant. Case is significant; uppercase and lowercase characters are distinct for identifiers. These characteristics apply to all identifiers, internal and external, in all TMS320 floating-point tools. (ANSI 3.1.2, K&R A2.3)
- The source (host) and execution (target) character sets are assumed to be ASCII. Although the compiler recognizes the syntax of multibyte characters, there are no additional multibyte characters. (ANSI 2.2.1, K&R A12.1)
- Hex or octal escape sequences in character or string constants may have values up to 32 bits. (ANSI 3.1.3.4, K&R A2.5.2)
- Character constants with multiple characters are encoded as the last character in the sequence. For example,

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

Data types

- For information about the representation of data types, refer to Section 3.2. (ANSI 3.1.2.5, K&R A4.2)
- The type size_t, which is assigned to the result of the sizeof operator, is equivalent to unsigned int. (ANSI 3.3.3.4, K&R A7.4.8)
- The type ptrdiff_t, which is assigned to the result of pointer subtraction, is equivalent to int. (ANSI 3.3.6, K&R A7.7)

Conversions

- ❑ Int-to-float conversions use the TMS320C3x/C4x FLOAT instruction which produces an exact representation. However, the value may be truncated when written to memory, resulting in a loss of precision towards negative infinity.

  (ANSI 3.2.1.3, K&R A6.3)

- ❑ Pointers and integers can be freely converted.

  (ANSI 3.3.4, K&R A6.6)

Expressions

- ❑ When two signed integers are divided and either is negative, the quotient is negative. A signed modulus operation takes the sign of the dividend (the first operand). For example,

  ```
  10 / -3 == -3,     -10 / 3 == -3
  10 % -3 == 1,      -10 % 3 == -1
  ```

  (ANSI 3.3.5, K&R A7.6)

- ❑ A right shift of a signed value is an arithmetic shift; that is, the sign is preserved. (ANSI 3.3.7, K&R A7.8)

Declarations

- ❑ The *register* storage class is effective for all character, short, integer, and pointer types. (ANSI 3.5.1, K&R A8.1)

- ❑ Structure members are not packed into words (with the exception of bit fields). Each member is aligned on a 32-bit word boundary.

  (ANSI 3.5.2.1, K&R A8.3)

- ❑ A bit field of type integer is signed. Bit fields are packed into words, beginning at the low-order bits, and do not cross word boundaries.

  (ANSI 3.5.2.1, K&R A8.3)

Preprocessor

- ❑ The preprocessor recognizes one #pragma directive; all others are ignored. For details, see Section 3.4, *Pragma Directives.* The recognized pragma is:

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

- ❑ The standard #error preprocessor directive forces the compiler to issue a diagnostic message and halt compilation. The TMS320 floating-point compiler extends the #error directive with a #warn directive, which, like #error, forces a diagnostic message but does not halt compilation. The syntax of #warn is identical to #error.

  (K&R A12.7)

3.2 Data Types

- All integer types (char, short, int, long, and their unsigned counterparts) are equivalent types and are represented as 32-bit binary values.
- Signed types are represented in 2s-complement notation.
- The type char is a signed type, equivalent to int.
- Objects of type enum are represented as 32-bit values; in expressions, the type enum is equivalent to int.
- All floating-point types (float, double, and long double) are equivalent and are represented in the TMS320C3x/C4x's 32-bit floating-point format.
- Although floating point types are not directly supported, support is provided for conversion to and from IEEE single and double-precision format for the TMS320C40 processor through the intrinsic functions TOIEEE and FRIEEE.

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

Table 3–1. TMS320 Floating-Point C Data Types

Type	Size	Representation	Range	
			Minimum	**Maximum**
char, signed char	32 bits	ASCII	–2147483648	2147483647
unsigned char	32 bits	ASCII	0	4294967295
short	32 bits	2s complement	–2147483648	2147483647
unsigned char	32 bits	binary	0	4294967295
int, signed int	32 bits	2s complement	–2147483648	2147483647
unsigned int	32 bits	binary	0	4294967295
long, signed long	32 bits	2s complement	–2147483648	2147483647
unsigned long	32 bits	binary	0	4294967295
enum	32 bits	2s complement	–2147483648	2147483647
float	32 bits	TMS320C3x/C4x	5.8774817e–39	3.4028235e38
double	32 bits	TMS320C3x/C4x	5.8774817e–39	3.4028235e38
long double	32 bits	TMS320C3x/C4x	5.8774817e–39	3.4028235e38
pointers	32 bits	binary	0	0xFFFFFFFF

Many of the range values are available as standard macros in the header file limits.h, which is supplied with the compiler.

Note: A TMS320C3x/C4x Byte Is 32 Bits

The ANSI definition specifies that the sizeof operator yields the number of **bytes** required to store an object. ANSI further stipulates that when the sizeof operator is applied to type char, the result is one. Since the TMS320C3x/C4x char is 32 bits (to make it separately addressable), a byte is also 32 bits. This yields results you may not expect; for example,
sizeof (int) == 1 (not 4).
TMS320C3x/C4x bytes and words are equivalent (32 bits).

3.3 Register Variables

The TMS320 floating-point C compiler treats register variables (variables declared with the *register* keyword) differently, depending on whether or not 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 maximize register use.

Compiling Without the Optimizer

If you use the register keyword, you can suggest variables as candidates for allocation into registers.

Eight (nine for the TMS320C4x) registers are available for variables in each function:

- Registers R4 and R5 (and R8 for the 'C4x) are reserved for the first two (or three) integer register variables in a function.
- Two registers, R6 and R7, are reserved for the first two floating-point register variables in a function.
- Four registers, AR4–AR7, are reserved for pointer register variables.

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

The register storage class is meaningful for parameters as well as local variables. A parameter declared as a register is passed on the stack normally but then moved into a register upon function entry. This improves access to the parameter within the function.

For more information about register variables, refer to Section 4.3 on page 4-11.

3.4 Pragma Directives

The TMS320 floating point C compiler supports the following pragma:

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

The DATA_SECTION pragma is useful if you have data objects that you would like to link into an area separate from the .bss.

Using the DATA_SECTION pragma, you could specify that data should go in another section, and then you could link that section into a different location in RAM with the linker.

The DATA_SECTION pragma will allocate space for the symbol in a section named *section name*. The section name should be a maximum of eight characters. The symbol must have file scope. That is, it cannot be defined or declared inside of a function body.

The pragma itself must be specified outside of a function body and must occur without any declaration, definition or reference to the symbol. If you do not follow these rules, the system will silently ignore the pragma and you will get a warning message.

Example 3–1. Pragma Directive Example

```
/*  Allocate space in section called ".mysection" for int f00. */
#pragma DATA_SECTION (f00, ".mysection")
int  f00;

int main ()
{
   bar = s;
   f00 = bar +1;
   return f00;
}
```

This code will create a section called .mysection and will allocate space within that section for the variable f00. All references to f00 will be indirect, because f00 is not in .bss.

3.5 The asm Statement

The TMS320 floating-point C compiler allows you to imbed TMS320C3x/C4x assembly language instructions or directives directly into the assembly language output of the compiler. This capability is provided through an extension to the C language: the *asm* statement. The asm statement is syntactically like a call to a function named asm, with 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:

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

The inserted code must be a legal assembly language statement. Like all assembly language statements, the line *must* begin with a label, a blank, a tab, or a comment (asterisk or semicolon). The compiler performs no checking on the string; if there is an error, it will be detected by the assembler. For more information, refer to the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide.*

These asm statements do not follow the syntactic restrictions of normal C statements: they can appear as either a statement or a declaration, even outside blocks. This is particularly useful for inserting directives at the very beginning of a compiled module.

Note: Avoid Disrupting the C Environment With asm Statements

Be extremely careful not to disrupt the C environment with asm statements. The compiler does not check the inserted instructions. Inserting jumps and labels into C code can cause unpredictable results in variables manipulated in or around the inserted code. Directives that change sections or otherwise affect the assembly environment can also be troublesome. Be especially careful when you use the optimizer with asm statements. Although the optimizer cannot remove asm statements (except where such statements are totally unreachable), it can significantly rearrange the code order near asm statements, possibly causing undesired results. The asm command is provided so that you can access features of the hardware, which, by definition, C is unable to access.

3.6 Initializing Static and Global Variables

The ANSI C standard specifies that static and global variables without explicit initializations must be specifically initialized to zero before the program begins running. This task is typically done when the program is loaded. Because the loading process depends heavily on the specific environment of the target application system, the TMS320 floating-point C compiler itself does not preinitialize variables; therefore, it is up to your application to fulfill this requirement.

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

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

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

Initializing Static and Global Variables With the const *Type Qualifier*

2/12/KOL
Getting Started
Guide : 2.4

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

const static and global variables without explicit initializations may or may not not be preinitialized to zero, for the same reasons discussed in Section 3.6. For example:

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

However, initialized const global and static variable are different because they are declared and initialized in a section called .const. For example:

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

which corresponds to an entry in the .const section:

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

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

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

The ANSI C language is a superset of the de facto C standard defined in the first edition of *The C Programming Language* (K&R). Most programs written for earlier non-ANSI C versions of the TMS320 floating-point C compiler and other non-ANSI compilers should correctly compile and run without modification.

However, there are subtle changes in the language that may affect existing code. Appendix C in K&R (second edition) summarizes the differences between ANSI C and the previous C standard, called K&R C.

To simplify the process of compiling existing C programs with the TMS320C3x/C4x ANSI C compiler, the compiler has a K&R option (–pk) that modifies some of the semantic rules of the language for compatibility with older code. In general, the –pk option relaxes requirements that are stricter for ANSI C than for previous C standards. The –pk option does not disable any new features of the language such as function prototypes, enumerations, initializations, or preprocessor constructs. Instead, –pk simply liberalizes the ANSI rules without revoking any of the features.

The specific effects of –pk are include:

- The integral promotion rules have changed regarding promoting an unsigned type to a wider, signed type. Under K&R, the result type was an unsigned version of the wider type; under ANSI, the result type is a signed version of the wider type. This affects options that perform differently when applied to signed or unsigned operands. Namely, comparisons, division (and mod), and right shift. In this example, assume that short is narrower than int:

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

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

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

  Even without –pk, a violation of this rule is a code-E (recoverable) error. An alternative to using –pk to work around this situation is to use –pe, which converts code-E errors to warnings.

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

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

- ANSI interprets file scope definitions that have no initializers as *tentative definitions*: in a single module, multiple definitions of this form are fused together into a single definition. Under K&R, each definition is treated as a separate definition, resulting in multiple definitions of the same object (and usually an error). For example:

  ```
  int a;
  int a;           /* illegal if -pk used, OK if not */
  ```

  Under ANSI, the result of these two declarations is a single definition for the object *a*. For most K&R compilers, this sequence is illegal because *a* is defined twice.

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

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

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

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

- ANSI specifies that bit fields must be of type integer or unsigned. With –pk, bit fields can be legally declared with any integer type. For example,

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

- K&R syntax allows assignments to structures returned from a function:

  ```
  f().x = 123         /* illegal unless -pk used */
  ```

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

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

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

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

3.8 Compiler Limits

Due to the variety of host systems supported by the TMS320 floating-point C compiler and the limitations of some of these systems, the compiler may not be able to successfully compile source files that are excessively large or complex. Most of these conditions occur during the first compilation pass (parsing). When such a condition occurs, the parser issues a code-I diagnostic message indicating the condition that caused the failure; usually the message also specifies the maximum value for whatever limit has been exceeded. The code generator also has compilation limits but fewer than the parser.

In general, exceeding any compiler limit prevents continued compilation, so the compiler aborts immediately after printing the error message. The only way to avoid exceeding a compiler limit is to simplify the program or parse and preprocess in separate steps.

Many compiler tables have no absolute limits but rather are limited only by the amount of memory available on the host system. Table 3–2 specifies the limits that are absolute. When two values are listed, the first is for PCs (DOS or OS/2), and the second is for other hosts. All the absolute limits equal or exceed those required by the ANSI C standard.

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.
- Use the protected-mode compiler

Table 3–2. Absolute Compiler Limits

Description	Limits	
Filename length	256 characters	
Source line length	2048 characters	(see note 1)
Length of strings built from # or ##	512 characters	(see note 2)
Macro definitions	Allocated from available system memory	
Macros predefined with –d	32	
Macro parameters	32	
Macro nesting	32 levels	(see note 3)
#include search paths	32 paths	(see note 4)
#include file nesting	32 levels	
Conditional inclusion (#if) nesting	32 levels	
Nesting of struct, union, or prototype declarations	20 levels	
Function parameters	32 parms	
Array, function, or pointer derivations on a type	12 derivations	
Aggregate initialization nesting	32 levels	
Static initializers	≈ 1500 per initialization	
Local initializers	≈ 150 per block	
Nesting of declarations in structs, unions, or protoypes	32 levels	
Global symbols	2000 10000	PCs All others
Block scope symbols visible at any point	500 1000	PCs All others
Number of unique string constants	400 1000	PCs All others
Number of unique floating-point constants	400 2000	PCs All others

Notes:
1) After splicing of \ lines. This limit also applies to any single macro definition or invocation.
2) Before concatenation. All other character strings are unrestricted.
3) Includes argument substitutions.
4) Includes –i and C_DIR directories.

Chapter 4

Runtime Environment

This section describes the TMS320 floating-point C runtime environment. To ensure successful execution of C programs, it is critical that all runtime code maintain this environment. If you write assembly language functions that interface to C code, follow the guidelines in this chapter.

Topics in this chapter include:

Topic **Page**

4.1 Memory Model

The C compiler treats memory as a single linear block that is partitioned into subblocks of code and data. Each block of code or data that a C program generates will be placed in its own contiguous space in memory. The compiler assumes that a full 32-bit address space is available in target memory.

Note: The Linker Defines the Memory Map

The **linker**, *not the compiler*, defines the memory map and allocates code and data into target memory. The compiler assumes nothing about the types of memory available, about any locations not available for code or data (holes), or about any locations reserved for I/O or control purposes. The compiler produces relocatable code that allows the linker to allocate code and data into the appropriate memory spaces. For example, you can use the linker to allocate global variables into fast internal RAM or to allocate executable code into internal ROM. Each block of code or data could be allocated individidually into memory, but this is not a general practice. (An exception to this is memory-mapped I/O, although physical memory locations can be accessed with C pointer types.)

4.1.1 Sections

The compiler produces six relocatable blocks of code and data; these blocks, called *sections*, can be allocated into memory in a variety of ways, to conform to a variety of system configurations. For more information about sections, please read the *Introduction to Common Object File Format* in the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide.*

There are two basic types of sections:

❑ **Initialized sections** contain data or executable code. The C compiler creates three initialized sections: .text, .cinit, and .const.

- The **.text section** is an initialized section that contains all the executable code as well as floating-point constants.
- The **.cinit section** is an initialized section that contains tables with the values for initializing variables and constants.
- The **.const section** is an initialized section that contains floating-point constants and switch tables.

❑ **Uninitialized sections** reserve space in memory (usually RAM). A program can use this space at runtime for creating and storing variables. The compiler creates three uninitialized sections: .bss, .stack, and .sysmem.

- The **.bss section** reserves space for global and static variables; in the small model, the .bss section also reserves space for the constant table. At program startup, the C boot routine copies data out of the .cinit section (which may be in ROM) and stores it in the .bss section.
- The **.stack section** allocates memory for the system stack. This memory is used to pass arguments to functions and to allocate local variables.
- The **.sysmem section** is a memory pool, or heap, used by the dynamic memory functions, malloc, calloc, and realloc. If a C program does not use these functions, the compiler does not create the .sysmem section.

The linker takes the individual sections from different models and combines sections with the sme name to create ouput sections. Note that the *assembler* creates three default sections (.text, .bss, and .data); the C compiler, however, does not use the .data section.

The complete program is made up of the compiler output sections, plus the assembler's .data section. The linker takes the individual sections from different modules and combines sections having the same name to create the output sections. You can place these output sections anywhere in the address space, as needed, to meet system requirements. The .text, .cinit, .const, and .data sections can be linked into either ROM and RAM. The .bss, .stack, and .sysmem sections should be inked into some type of RAM. Note, however, that the .bss and .const sections must be allocated in the same 64K data page for a small model, see page 4-5.

For more information about allocating sections into memory, see the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide.*

4.1.2 C System Stack

The C compiler uses a software stack to:

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

The runtime stack grows up from low addresses to higher addresses. The compiler uses two auxiliary registers to manage this stack:

SP is the **stack pointer** (SP); it points to the current top of the stack.

AR3 is the **frame pointer (FP)**; it points to the beginning of the current frame. Each function invocation creates a new frame at the top of the stack, from which local and temporary variables are allocated.

The C environment manipulates these registers automatically. If you write any assembly language routines that use the runtime stack, be sure to use these registers correctly. (For more information about using these registers, see Section 4.3, page 4-11; for more information about the stack, see Section 4.4, page 4-16.)

The stack size is set by the linker. The linker also creates a global symbol, __STACK_SIZE, and assigns it a value equal to the size of the stack in words. The default stack size is 1K (400h) words. This size allows the stack to fit into one of the on-chip RAM blocks. You can change the size of the stack at link time by using the –stack option on the linker command line and specifying the size of the stack as a constant immediately after the option. For more information about the –stack option, refer to the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide.*

At system initialization, the SP is set to a designated address for the bottom-of-stack. This address is the first location in the .stack section. Since the position of the stack depends on where the .stack section is allocated, the actual position of the stack is determined at link time. If you allocate the stack as the last section in memory (highest address), the stack has unlimited space for growth (within the limits of system memory).

Note: Stack Overflow

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

4.1.3 Dynamic Memory Allocation

The runtime-support library supplied with the compiler contains several functions (such as malloc, calloc, and realloc) that allow you to dynamically allocate memory for variables at runtime. Dynamic allocation is not a standard part of the C language; it is provided by standard runtime-support functions.

Memory is allocated from a global pool, or heap, that is defined in the .sysmem section. You can set the size of the .sysmem section by using the –heap option when you link your program. Specify the size of the memory pool as a constant after the –heap option on the linker command line. The linker also creates a global symbol, __SYSMEM_SIZE, and assigns it a value equal to the size of the heap in words. The default size is 1K words. For more information on the heap option, refer to the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide.*

Dynamically allocated objects are not addressed directly (they are always accessed with pointers), and the memory pool is in a separate section; therefore, the dynamic memory pool can have an unlimited size, even in the small memory model. This allows you to use the more efficient small memory model, even if you declare large data objects. To conserve space in the .bss, you can allocate large arrays from the heap instead of declaring them as global or static. For example, instead of a declaration such as:

```
struct big table[10000];
```

use a pointer and call the malloc function;

```
struct big *table
table = (struct big *)malloc(10000 * sizeof (struct big));
```

4.1.4 Big and Small Memory Models

The compiler supports two memory models that affect how .bss is allocated into memory. Neither model restricts the size of the .text or .cinit sections. Both models restrict the size of a single function to 32K words of code or less; this allows the compiler to use relative conditional jumps over the entire range of the function.

- **The small memory model,** which is the default, requires that the entire .bss section fit within a single 64K-word memory data page (65536 words). This means that the total space for all static and global data in the program must be less than 64K and that the .bss cannot cross any 64K address boundaries. The compiler sets the data-page pointer register (DP) during runtime initialization to point to the beginning of the .bss. Then the compiler can access all objects in the .bss (global and static variables and constant tables) with direct addressing (@) without modifying the DP.

- **The big memory model** does not restrict the size of the .bss; unlimited space is available for global and static data. However, when the compiler accesses any global or static object that is stored in the .bss, it must first ensure that the DP correctly identifies the memory page where the object is stored. To accomplish this, the compiler must set the DP by using an LDP or LDPK (load data-page pointer) instruction each time a static or global data item is accessed. This task produces one extra instruction word and several extra machine cycles (one cycle for the LDP instruction plus one or more pipeline delay cycles if the object is accessed by the next instruction).

For example, the following compiler-generated assembly language uses the LDP instruction to set the DP to the correct page before accessing the global variable x. This is a TMS320C3x example (the sequence is one cycle shorter for the 'C4x):

```
*** _x is a global variable ***

       LDP    _x         ; 1 extra word, 1 cycle
       LDI    @_x,R0     ; 3 cycles (2 pipeline delays)
```

To use the big memory model, invoke the compiler with the –mb option. For more information, refer to Section 2.3, *Using Runtime Models*, on page 2-29.

4.1.5 RAM and ROM Models

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

In situations where a program is loaded directly from an object file into memory and then run, you can avoid having the .cinit section occupy space in memory. A user-defined loader can read the initialization tables directly from the object file (instead of from ROM) and perform the initialization directly at load time (instead of at runtime). You can specify this *to the linker* by using the –cr linker option. For more information, refer to Section 4.8, *System Initialization*, on page 4-33.

4.2 Object Representation

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

4.2.1 Data Type Storage

- All basic types are 32 bits wide and stored in individual words of memory. No packing is performed except on bit fields, which are packed into words. Bit fields are allocated from LSB (least significant bit) to MSB in the order in which they are declared.
- No object has any type of alignment requirement; any object can be stored on any 32-bit word boundary. Objects that are members of structures or arrays are stored just as they are individually. Members are not packed into structures or arrays (unless the members are bit fields).
- The integer types *char, short, int,* and *long* are all equivalent, as are their unsigned counterparts. Objects of type *enum* are also represented as 32-bit words.
- The *float* and *double* types are equivalent; both types specify objects represented in the TMS320C3x/C4x's 32-bit floating-point format.

For more information on data types, refer to Section 3.2, *Data Types*, on page 3-4.

4.2.2 Long Immediate Values

The TMS320C3x/C4x instruction set has no immediate operands that are longer than 16 bits. The compiler occasionally needs to use constants that are too long to be immediate operands. This occurs with signed integer constants that have more than 15 significant nonsign bits, with unsigned integers that have more than 16 significant bits, or with floating-point constants that have more than 11 significant nonsign bits in the mantissa.

4.2.3 Addressing Global Variables

The compiler generates the addresses of global and static symbols for indexing arrays or manipulating pointers. Because these addresses may be up to 32 bits wide and immediate operands are limited to 16 bits, these addresses are treated like long constants. Subsection 4.2.5 on page 4-9 describes the structure of the constant table.

4.2.4 Character String Constants

In C, a character string constant can be used in either of two ways:

- ❑ It can initialize an array of characters; for example:

  ```
  char s[ ] = "abc";
  ```

  When a string is used as an initializer, it is simply treated as an initialized array; each character is a separate initializer. For more information about initialization, refer to Section 4.8, *System Initialization*, on page 4-33.

- ❑ It can be used in an expression; for example:

  ```
  strcpy (s, "abc");
  ```

  When a string is used in an expression, the string itself is defined in the .const section with the .byte assembler directive, along with a unique label that points to the string (the terminating 0 byte is also included). In the following example, the label SL5 points to the string from the example above:

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

  String labels have the form **SLn**, n being a number assigned by the compiler to make the label unique. These numbers begin at 0 with an increase of 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 SLn represents the address of the string constant. The compiler uses this label to reference the string in the expression. Like all addresses of static objects, this address must be stored in the constant table in order to be accessed. Thus, in addition to storing the string itself in the .const section, the compiler uses the following directive statement to store the string's address in the constant table:

  ```
          .word SLn
  ```

  If the same string is used more than once within a source module, the string will not be duplicated in memory. All uses of an identical string constant share a single definition of the string.

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

  ```
  char   *a = "abc"
  a[1] = 'x';        /* Incorrect!   */
  ```

4.2.5 The Constant Table

The constant table contains definitions of all the objects that the compiler must access, but are too wide to be used as immediate operands. Such objects include:

- integer constants that are wider than 16 bits
- floating-point constants that have exponents larger than 4 bits or mantissas larger than 11 bits
- addresses of global variables
- addresses of string constants

The constant table is simply a block of memory that contains all such objects. The compiler builds the constant table at the end of the source module by using the .word and .float assembler directives. Each entry in the table occupies one word. The label CONST is the address of the beginning of the table. For example:

```
CONST: .word  011223344h   ; 32 bit constant
       .float 3.14159265   ; floating-point
       .word  _globvar     ; address of global
       .word  SL23         ; address of string
```

Objects in the table are accessed with direct addressing; for example:

```
   LDI    @const+offset, R0
```

In this example, offset is the index into the constant table of the required object. As with string constants, identical constants used within a source module share a single entry in the table.

In the big memory model, the constant table is built in the .const section (and is not copied into RAM). The compiler must insure that the DP register is correctly loaded before accessing the object in the table, just as with accessing global variables. This requires an LDP instruction before each access of the constant table.

The small model, however, avoids the overhead of loading the DP by requiring that all directly addressable objects, including all global variables as well as the constant table, are stored on the same memory page. Of course, global variables must be stored in RAM. For the code to be ROM-able, the constant table must be in ROM. In order to get them on the same page, the boot routine must copy the constant table from permanent storage in ROM to the global page in RAM. The compiler accomplishes this by placing the data for the constant table in the .cinit section and allocating space for the table itself .bss. Thus the table is automatically built into RAM through the autoinitialization process.

[illegible]: Getting Started Guide: 2.4

Note that the small memory model restricts the total size of the global data page, including the constant table and .const section, to 64K words.

As with all autoinitialization, you can avoid the extra use of memory required by the .cinit section by using the –cr linker option and a “smart” loader to perform the initialization directly from the object file. For more information about autoinitialization, refer to Section 4.8 on page 4-33.

4.3 Register Conventions

Strict conventions associate specific registers with specific operations in the C environment. If you plan to interface an assembly language routine to a C program, it is important that you understand these register conventions.

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

The compiler uses registers differently, depending on whether or not you use the optimizer (–o option). The optimizer uses additional registers for register variables (variables defined to reside in a register rather than in memory). *However, the conventions for preserving registers across function calls are identical with or without the optimizer.*

The following table summarizes how the compiler uses the TMS320C3x/C4x registers and shows which registers are defined to be preserved across function calls.

Table 4–1. Register Use and Preservation Conventions

Register	Use Without the Optimizer	Use With the Optimizer	Preserved by Call
R0	Integer and float expressions Scalar return values	Integer and float expressions Scalar return values	No No
R1	Integer and float expressions	Integer and float expressions	No
R2–R3	Integer and float expressions	Integer and float register variables	No
R4–R5	Integer register variables	Integer and float register variables Float register variables	Integer part preserved
R6–R7	Float register variables	Integer and float register variables Integer register variables	Floating part preserved
AR0–AR1	Pointer expressions	Pointer expressions	No
AR2	Pointer expressions	Integer and pointer register variables	No
AR3	Frame ointer (FP)	Frame Pointer (FP)	Yes
AR4–AR7	Pointer register variables	Pointer register variables	Yes
IR0	Extended frame offsets	Extended frame offsets	No
IR1	Extended frame offsets	Integer register variables	No
BK	Not used	Integer register variables	No
RC, RS, RE	Block (structure) copy	Block (structure) copy Block repeat loops Integer register variables	No
SP	Stack pointer	Stack Pointer	Yes
DP	Accessing globals (big model only)	Accessing globals (big model only)	Yes in small model No in big model
TMS320C4x Only			
R8	Integer register variables	Integer register variables Float register variables	Integer part preserved
R9–R11	Integer and float variables	Integer and float variables	No

4.3.1 Register Variables

Register variables are local variables or compiler temporaries defined to reside in a register rather than in memory. Storing local variables in registers allows significantly faster access, which improves the efficiency of compiled code. Table 4–2 shows the registers that are reserved to store only register variables.

Table 4–2. Registers Reserved for Register Variables

Register	Description
R4, R5	Integer register variables
R6, R7	Floating-point register variables
R8	('C4x only) Integer register variables
AR4–AR7	Pointer register variables

These registers are preserved across calls. When a function uses register variables, it must save the contents of each used register on entrance to the function, then restore the saved contents on exit. This ensures that a called function does not disrupt the register variables of the caller.

When you are not using the optimizer (–o option), you can allocate register variables with the *register* keyword. Registers are assigned to variables in the order that they are declared; for example, the first integer variable declared as register is assigned to R4, and the second to R5. If a function declares more register variables than are available for that type, the excess variables are treated as automatic variables and are stored in memory on the local frame.

When you are using the optimizer, the compiler ignores the register keyword and uses a cost analysis algorithm to allocate variables and temporaries to registers. The optimizer can also allocate variables to registers that are not dedicated for use as variable registers. Table 4–3 shows the additional registers that are available to the optimizer.

Table 4–3. Additional Registers Available to the Optimizer for Register Variables

Register	Description
R2, R3, R6, R7, AR2, IR1, BK, RC, RS, RE	Additional integer register variables
R2, R3, R4, R5	Additional floating-point register variables
R9, R10, R11	('C4x only) additional integer or floating-point register variables
AR2	Additional pointer register variables

Note: Some registers appear in more than one table. This indicates that the register is available for use in more than one way; a register may be used for only one variable at at a time.

These additional registers are not preserved across function calls, so they are used only for variables that do not overlap any calls.

Because the register use for variables depends on whether or not you use the optimizer, you should avoid writing code that depends on specific register variables allocated to specific registers.

4.3.2 Expression Registers

The compiler uses registers not being used for register variables to evaluate expressions and store temporary results. The compiler keeps track of the current contents of the registers and attempts to allocate registers for expressions in a way that preserves the useful contents in the registers whenever possible. This allows the compiler to reuse register data, to take advantage of the TMS320C3x/C4x's efficient register addressing modes, and to avoid unnecessary accesses of variables and constants.

The contents of the expression registers are not preserved across function calls. Any register that is being used for temporary storage when a call occurs is saved to the local frame before the function is called. This prevents the called function from ever having to save and restore expression registers.

If the compiler needs another register for storing an expression evaluation, a register that is being used for temporary storage can be saved on the local frame and used for the expression analysis. Typical expressions seldom require more than four expression registers.

4.3.3 Return Values

When a value of any scalar type (integer, pointer, or floating point) is returned from a function, the value is placed in register R0 when the function returns.

4.3.4 Stack and Frame Pointers

The TMS320 floating-point C compiler uses a conventional stack mechanism for allocating local (automatic) variables and passing arguments to functions. When a function requires local storage, it creates its own working space (local frame) from the stack. The local frame is allocated during the function's entry sequence and deallocated during the return sequence.

Two registers, the stack pointer (SP) and the frame pointer (FP), manage the stack and the local frame. The SP is a TMS320C3x/C4x register dedicated to managing the stack. The compiler uses SP in the conventional way: the stack grows toward higher addresses, and the stack pointer points to the top location (highest memory) on the stack. Register AR3 is dedicated as the frame pointer (FP). The FP points to the beginning or bottom of the local frame for the current function. All objects stored in the local frame are referenced indirectly through the FP.

Both the FP and the SP must be preserved across function calls. The function calls automatically preserve the SP because everything pushed for the call is popped on return. The FP is preserved as a specific part of a function's entry and exit sequence. For more information about stack and frame pointer use during function calls, refer to Section 4.4, *Function Structure and Calling Connections*, page 4-16.

4.3.5 Other Registers

Other registers that have specific functions are discussed below.

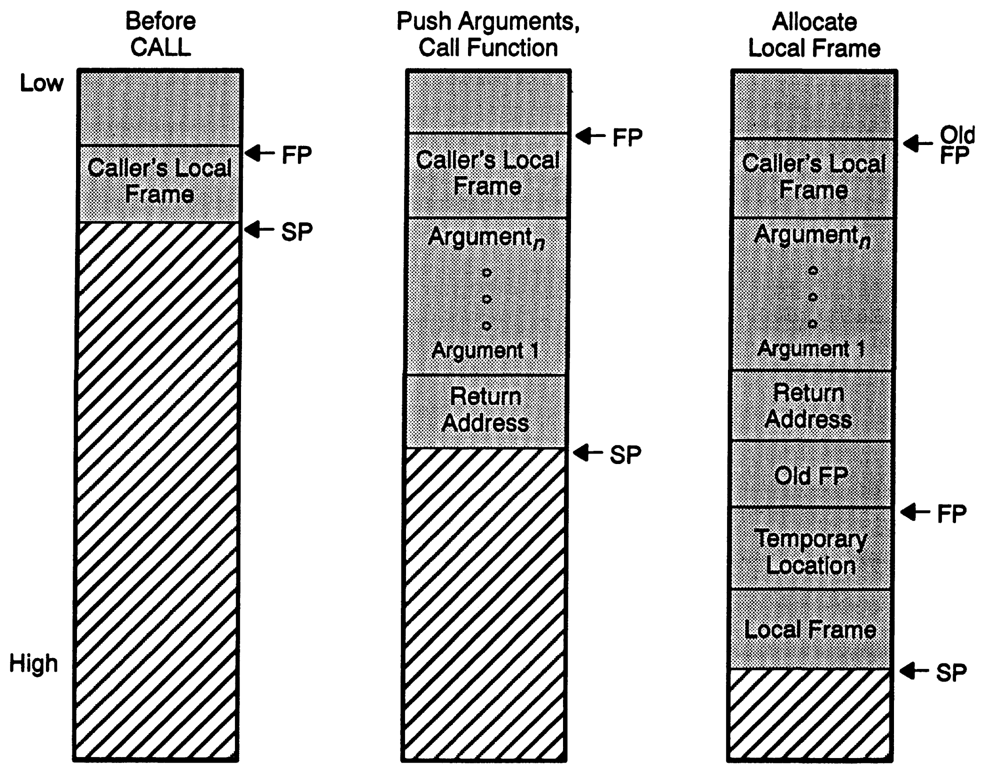
- The data-page pointer (DP) is used to access global and static variables. In the small model, the DP is set at program startup and never changed. If the DP is modified by an assembly language function in the small model, the function must preserve the value.
- Index registers IR0 and IR1 are used for array indexing and when an offset of more than 8 bits (±255 words) is required for indirect addressing. In addition, the optimizer uses IR1 for a general-purpose integer register variable. Neither register is preserved across calls.
- The BK register is used only by the optimizer as an integer register variable. Its value is not preserved across calls.
- The compiler uses block-repeat registers (RC, RE, and RS) to generate efficient block copy operations for assigning large (>5 words) structures. The optimizer also uses these registers for program loops or for integer register variables. The repeat register values are not preserved across calls.

4.4 Function Structure and Calling Conventions

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

Figure 4–1 illustrates a typical function call. In this example, parameters are passed to the function, and the function uses local variables. If you use the register-argument model, see Section 4.4.2, some or all of the arguments may be passed in registers rather than on the stack as shown. This example also shows allocation of a local frame for the called function. Functions that have *no* arguments passed on the stack *and no* local variables do not allocate a local frame.

Figure 4–1. Stack Use During a Function Call



4.4.1 Function Call, Standard Runtime Model

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

1) The caller pushes the arguments on the stack in reverse order (the right-most declared argument is pushed first, and the leftmost is pushed last). This places the leftmost argument at the top of the stack when the function is called.

2) The caller calls the function.

3) When the called function is complete, the caller pops the arguments off the stack with the following instruction:

   `SUBI n,SP` (*n* is the number of argument words that were pushed)

4.4.2 Function Call, Register Argument Runtime Model

1) Six registers, AR2, R2, R3, RC, RS, and RE, are used to pass arguments as follows:

   - The first two (left-most) floating-point (float, double, or long double) arguments are passed in R2 and R3.
   - The remaining integer or pointer arguments are passed in the remaining registers in the above list, in order.
   - All arguments that are not assigned to registers, either because of type incompatibility or because all registers have been exhausted, are pushed onto the stack in reverse order (right-to-left).

   In Figure 4–2, several function prototypes are shown with the location of each argument indicated to illustrate the conventions.

Figure 4–2. Register Argument Conventions

```
int fl(int *a, int b, int c);   /* function call */
          AR2     R2      R3        <——where parameters are placed

int f2(int a, float b, int *c, struct A d, float e, Int f); /* call */
         AR2      R2       RC        STACK       R3       RS   <——parameters

int f3(float a, int *b, float c, int d, float e);   /* call */
          R2       AR2      R3       RC     STACK  <——parameters

int f4(struct x a, int b, int C, int d, int e, int f, int g, int h);
            STACK    AR2     R2     R3     RC     RS     RE   STACK
```

Notice how R2 and R3 are allocated first to any floats in the argument list. Then, a second pass allocates remaining arguments to remaining registers.

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

```
int vararg(int a, int b, ...);
```

AR2 STACK STACK STACK STACK...

2) The caller calls the function.

3) The caller pops the arguments that were passed on the stack, if any.

4.4.3 Responsibilities of a Called Function

A called function must perform certain tasks. Step 1 below helps to manage the local frame. If the function has no local variables, no stack arguments, and no need for local temporary storage, Steps 1) and 6) are not taken.

1) The called function sets up the local frame. The local frame is allocated in the following way:
   a) The old frame pointer is saved on the stack.
   b) The new frame pointer is set to the current SP.
   c) The frame is allocated by adding its size to the SP.

2) If the called function modifies any of the following registers, it must save them on the stack.

Save as Integers		Save as floating-point	
R4	R5	R6	R7
AR4	AR5		
AR6	AR7		
FP	DP (small model only)		
	R8 ('C4x only)		

The called function may modify any other registers without saving them.

3) The called function executes its code.

4) If the function returns an integer, pointer, or float, it places the return value in R0.

   If the function returns a structure, the caller allocates space for the structure and then passes the address of the return space to the called function in register AR0. To return a structure, the called function then copies the structure to the memory block that AR0 points to. If the caller does not use the return value, AR0 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 place the address of s in AR0 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 AR0) and where they are defined (so the function knows to copy the result).

5) The called function restores all saved registers.

6) It deallocates the local frame (if necessary) by subtracting its size from SP and restores the old FP by popping it.

7) It pops the return address and branches to it. In functions with no saved registers, this is performed by executing a RETS statement. In other functions, the compiler loads the return address into R1 and uses a delayed branch to return. The delay slots are used to restore registers and deallocate the local frame.

4.4.4 Accessing Arguments and Local Variables

A function accesses its stack arguments and local nonregister variables indirectly through the FP, which always points to the bottom of the local frame. Because the FP actually points to the old FP, the first local variable is addressed as *+FP(1). Other local variables are addressed with increasing offsets, up to a maximum of 255. Local objects with offsets larger than 255 are accessed by first loading their offset into an index register (IR*n*) and addressing them as *+FP(IR*n*).

Stack arguments are addressed in a similar way, but with negative offsets from the FP. The return address is stored at the location directly below the FP, so the first argument is addressed as *–FP(2). Other arguments are addressed with increasing offsets, up to a maximum of 255 words. The IR registers are also used to access arguments with offsets larger than 255.

Note: Avoid Locals and Arguments With Large Offsets

It is desirable to avoid using locals and arguments with offsets larger than 255 words. However, if you must use locals and/or arguments with large offsets, the sequence used to access these variables is:

```
LDI    offset, IRn
...    *+FP(IRn), ...
```

This sequence incurs one additional instruction and three additional clock cycles each time it is used. If you must use a larger local frame, try to put the most frequently used variables within the first 255 words of the frame.

When register arguments are used, it is critically important that the caller and callee agree both in type and number of arguments so that the called function can find the arguments. Rules for argument passing are described in subsection 4.4.2. Arguments passed in registers can remain in those registers as long as the register is not needed for expression evaluation. If registers containing arguments must be freed for expression evaluation, they can be copied to the local frame or other registers at function entry.

When you are not using the optimizer, the compiler needs the argument-passing registers for expression evaluation, so the arguments are copied to a local frame at function entry.

When you are using the optimizer, the compiler attempts to keep register arguments in their original registers.

4.5 Interfacing C With Assembly Language

There are three ways to use assembly language in conjunction with C code:

- Use separate modules of assembled code and link them with compiled C modules (see subsection 4.5.1, page 4-21). This is the most versatile method.
- Use inline assembly language, embedded directly in the C source (see subsection 4.5.3, page 4-27).
- Modify the assembly language code that the compiler produces.

Note: Assembler Support for Runtime Models

The two argument-passing runtime models use different function structure and calling conventions. Assembly language functions called by C need to retrieve arguments according to the runtime model that was used to compile the code. The compiler and assembler provide support for the two runtime models in the form of two predefined symbols. The assembler symbol .REGPARM is set to 1 when –mr is used and is set to 0 otherwise. The compiler symbol _REGPARM is defined to be 1 if –mr is used and is set to 0 otherwise. Example 4–1 on page 4-24 shows how to use these symbols to write code that will work with either runtime model.

4.5.1 Assembly Language Modules

Interfacing with assembly language functions is straightforward if you follow the calling conventions defined in Section 4.4 and the register conventions defined in Section 4.3. C code can access variables and call functions defined in assembly language, and assembly code can access C variables and call C functions. Follow these guidelines to interface assembly language and C:

- All functions, whether they are written in C or assembly language, must follow the conventions outlined in Section 4.3, page 4-11.
- You must preserve any dedicated registers modified by a function. (You must preserve the DP in the small model only.) These are the dedicated registers:

Save as integers		**Save as floating-point**	
R4	R5	R6	R7
AR4	AR5		
AR6	AR7		
FP	DP (small model only)		
SP	R8 (‘C4x only)		

sted under number 2 of this listAll registers are saved as integers except R6 and R7, which are saved as floating-point values. 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 is popped back off before the function returns (thus preserving SP).

All other registers (such as expression registers, index registers, status registers, and block repeat registers) are not dedicated and can be used freely without first being saved.

- Interrupt routines must save the registers they use. Registers R0–R7 (R0–R11 on the TMS320C4x) must be saved as complete 40-bit values because they may contain either integers or floating–point values when the interrupt occurs. For more information about interrupt handling, refer to Section 4.6 on page 4-29.

- Access arguments from the stack or in the designated registers, depending on which argument-passing model is used.

- When calling a C function from assembly language, follow the guidelines in subsection 4.4.1, on page 4-17; push the arguments on the stack in reverse order. When using the register-argument model, load the designated registers with arguments, and push the remaining arguments on the stack as described in subsection 4.4.2, on page 4-17. When calling C functions, remember that only the dedicated registers are preserved. C functions can change the contents of any other register.

- Functions must return values correctly according to their C declarations. Integers, pointers, and floating-point values are returned in register R0, and structures are returned as described in step 4 on page 4-18.

- No assembly module should use the .cinit section for any purpose other than autoinitialization of global variables. The C startup routine in boot.asm assumes that the .cinit section consists entirely of initialization tables. Disrupting the tables by putting other information in .cinit can cause unpredictable results.

- The compiler appends an underscore (_) to the beginning of all identifiers. In assembly language modules, you must use a prefix of _ for all objects that are to be accessible from C. For example, a C object named x is called _x in assembly language. For identifiers that are to be used only in an assembly language module or modules, any name that does not begin with a leading underscore may be safely used without conflicting with a C identifier.

- ❑ Any object or function declared in assembly that is to be accessed or called from 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 function or object from assembly, declare the C object with .global. This creates an undefined external reference that the linker will resolve.

Predefined Symbols

The assembler has several predefined symbols that allow you to write assembly code that is compatible with the different runtime models. You can conditionally assemble code corresponding to different models by using these symbols as control switches when you invoke the assembler with the appropriate options (this happens automatically when you use cl30):

Symbols	**Value**	**Description**
.TMS320C30	1 or 0	1 if –v30 option used
.TMS320C40	1 or 0	1 if –v40 option used
.REGPARM	1 or 0	1 if –mr option used
.BIGMODEL	1 or 0	1 if –mb option used

An Example of an Assembly Language Function

Example 4–1 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. Note that this example can be used with either of the argument-passing conventions.

In the assembly language code in Example 4–1, note the underscores on all the C symbol names. Note also how the DP must be set only when accessing global variables in the big model. For the small model, the LDP instruction can be omitted. To use this example with either the large or small model, use the predefined symbol .BIGMODEL to conditionally assemble the LDP statement.

Example 4–1.An Assembly Language Function Called From C

(a) C Program

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

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

(b) Assembly Language Program

```
   FP .set    AR3          ; FP is AR3

   .global _asmfunc        ; Declare external function
   .global _gvar           ; Declare external variable

_asmfunc:

   .if .REGPARM = 0        ; standard runtime model
   PUSH   FP               ; Save old FP
   LDI    SP,FP            ; Point to top of stack
   LDI    *–FP(2),AR2      ; Load argument into AR2
   .endif                  ; Both runtime models
   .if .BIGMODEL
   LDP    _gvar            ; Set DP to page of gvar
   .endif                  ;(BIG MODEL ONLY)
   LDI    @_gvar, RO
   ADDI   AR2,RO           ; RO = gvar + argument
    if .REGPARM = 0        ; Standard runtime model
   POP    FP               ;Restore FP
   .endif
   RETS
```

In the C program in Example 4–1, the *extern* declaration of asmfunc is optional because the function returns an int. Like C functions, assembly functions need be declared only if they return noninteger values. If you are using the register-argument runtime model, all functions, including assembly language functions, should have prototypes so that the arguments are passed correctly.

4.5.2 Accessing Assembly Language Variables From C

It is sometimes useful for a C program to access variables or constants defined in assembly language. There are three different 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.

Accessing Variables Defined in the .bss Section

Accessing uninitialized variables from the .bss section is straightforward:

- ❑ Use the .bss directive to define the variable.
- ❑ Use the .global directive to make the definition external.
- ❑ Remember to precede the name with an underscore.
- ❑ In C, declare the variable as *extern*, and access it normally.

Example 4–2 shows an example that accesses a variable defined in .bss.

Example 4–2. Accessing a Variable Defined in .bss From C

(a) Assembly Language Program

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

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

(b) C Program

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

Accessing Variables Not Defined in the .bss Section

2KOK : Started ting ule : 2 4

If a variable is not defined in the .bss section, it is more difficult to access it from C. A common situation is a lookup table, defined in assembly, that you don't want to put in RAM. In this case, you must define a pointer to the object and access it indirectly from C.

The first step is to define the object. You can define the object in its own section (either initialized or uninitialized), or you can define it in an existing section. After defining the object, declare a global label that points to the beginning of the object.

If the object has its own section, it can easily be linked anywhere into the memory space (this task is a little more difficult if the object is *sharing* an existing section). To access it in C, you must declare an additional C variable to point to the object. Initialize the pointer with the assembly language label declared for the object; remember to remove the underscore.

Note: Use the –mf Option When Using the Optimizer

The optimizer, by default, assumes that all variables are defined in the .bss section. As a result of this assumption, the optimizer is free to change indirect memory accesses to direct ones if it is more efficient to do so. If you are accessing a variable that is not defined in .bss, this assumption is invalid. Use the –mf option to force the optimizer to preserve indirect memory access.

Example 4–3 shows an example for accessing a variable that is not defined in .bss.

Example 4–3. Accessing a Variable Not Defined in .bss From C

(a) Assembly Language Program

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

(b) C Program

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

A reference such as sine[4] will not work because the object is not in .bss and because a direct reference such as this generates incorrect code.

Accessing Assembly Language Constants

You can define global constants in assembly language by using the .set and .global directives or in a linker command file using a linker assignment statement. These constants are accessible from C, but it is not straightforward.

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

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

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

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

(a) Assembly Language Program

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

(b) C Program

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

Because you are referencing only the symbol's value as stored in the symbol table, the symbol's declared type is unimportant. In Example 4–4, int is used. You can reference linker-defined symbols in a similar manner.

4.5.3 Inline Assembly Language

Within a C program, you can use the *asm statement* to inject a single line of assembly language into the assembly language file that the compiler creates. A series of asm statements places sequential lines of assembly language into the compiler output with no intervening code.

The asm statement is provided so that you can access features of the hardware that would be otherwise inaccessible from C. For example, you can modify the interrupt control registers to enable or disable interrupts. You can also access the status, interrupt enable, and interrupt flag registers.

Note: Avoid Disrupting the C Environment With asm Statements

Be extremely careful not to disrupt the C environment with asm statements. The compiler does not check the inserted instructions. Inserting jumps and labels into C code can cause unpredictable results in variables manipulated in or around the inserted code. Directives that change sections or otherwise affect the assembly environment can also be troublesome. Be especially careful when you use the optimizer with asm statements. Although the optimizer cannot remove asm statements, it can significantly rearrange the code order near asm statements, possibly causing undesired results. The asm command is provided so that you can access features of the hardware, which, by definition, C is unable to access.

The asm statement is also useful for inserting comments in the compiler output; simply start the assembly code string with an asterisk (*) as shown below:

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

4.6 Interrupt Handling

Interrupts can be handled directly with C functions by using a special naming convention:

c_int*nn*

nn is a two-digit interrupt number between 00 and 99. For example:

c_int01

Using one of these function names defines an interrupt routine. When the compiler encounters one of these function names, it generates code that allows the function to be activated from an interrupt trap.

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

4.6.1 Saving Registers During Interrupts

When C code or assembly code is interrupted, all registers must be preserved, including the status registers. A problem arises with the extended–precision registers R0–R7 (R0–R11 on the 'C4x): these registers can contain either integer or floating-point values, and an interrupt routine cannot determine the type of value in a register. Thus, an interrupt routine must preserve *all 40 bits* of any of these registers that it modifies. This involves saving both the integer part (lower 32 bits) and the floating-point part (upper 32 bits).

The following code illustrates the entry and exit sequences for an interrupt service routine that has two local variables and uses registers FP, SP, ST, R3, R4, and AR4:

error:
Getting Started
[illegible]

```
PUSH   ST     ; save ST
PUSH   FP     ; save old FP
LDI    SP,FP  ; set up new FP
ADDI   2,SP   ; allocate local frame
PUSH   R3     ; save lower 32 bits of R3
PUSHF  R3     ; save upper 32 bits of R3
PUSH   R4     ; save lower 32 bits of R4
PUSHF  R4     ; save upper 32 bits of R3
PUSH   AR4    ; save AR4
```

Exit

```
POP     AR4     ; restore AR4
POPF    R4      ; restore upper 32 bits of R4
POP     R4      ; restore lower 32 bits of R4
POPF    R3      ; restore upper 32 bits of R3
POP     R3      ; restore lower 32 bits of R3
POP     ST      ; restore ST
SUBI    2,SP    ; deallocate local frame
POP     FP      ; restore FP
RETI            ; return from interrupt
```

Notice how the upper and lower bits of R3 and R4 are saved and restored separately. Any extended-precision register must be saved in two parts. All other registers can be saved as integers.

4.6.2 Assembly Language Interrupt Routines

Interrupts can also be handled with assembly language code, as long as the register conventions are followed. Like *all* assembly functions, interrupt routines can use the stack, access global C variables, and call C functions normally. When calling C functions, be sure that all nondedicated registers are preserved because the C function can modify any of them. Of course, dedicated registers need not be saved because they are preserved by the C function. Interrupt handler functions, whether in C or assembly, must be installed by placing their address in the interrupt vector table. On the TMS320C3x, this table is located in the first 64 words of the address space. On the TMS320C4x, the table can be located anywhere on any 512 word boundary in the address space. The IVTP register points to the location of the interrupt vector table.

4.7 Runtime-Support Arithmetic Routines

The runtime-support library contains a number of assembly language functions that provide arithmetic routines for C math operators that the TMS320C3x/C4x instruction set does not provide, such as division. There are as many as three different versions of each of these functions. The versions are distinguished from each other by the number appended to the function name. For example, DIV_F30 is the version of DIV_F that is called when the code is compiled for the TMS320C3x, DIV_F40 is for the 'C4x, and DIV_F is the version called by code compiled by version 4.10 (and all earlier versions) of the compiler.

The TMS320C3x MPYI (multiply integer) instruction does not perform full 32-bit multiplication (the TMS320C4x MPYI instruction *does*); it uses only the lower 24 bits of each operand. Because standard C requires full 32-bit multiplication, a runtime-support function, MPY_I30, is provided to implement 32-bit integer multiplication for the TMS320C3x. This function does not follow the standard C calling sequence; instead, operands are passed in registers R0 and R1. The 32-bit product is returned in R0.

The compiler uses the TMS320C3x MPYI instruction only in cases where address arithmetic is performed (such as during array indexing); because no address can have more than 24 bits, a 24 x 24 multiply is sufficient. You can use the –mm option to force the compiler to use MPYI instructions for all integer multiplies.

The TMS320C4x MPYI instruction performs full 32-bit multiplication. Therefore, MPYI is always used when the –v40 option is selected and there is no MPY_I40 function.

Because the TMS320C3x/C4x has no division instructions, integer and floating-point division are performed via calls to additional runtime-support functions named DIV_I30 and DIV_F30 (or DIV_I40 and DIV_F40 for the TMS320C4x). Another function, MOD_I30, performs the integer modulo operation. Corresponding functions named DIV_U30 and MOD_U30 are used for unsigned integer division and modulo. Like MPY_I30, these functions take their arguments from R0 and R1 and return the result in R0, R1, R10 ('C4x only), AR0, and AR1. These functions differ for the TMS320C3x and the TMS320C4x because the 'C4x division functions take advantage of the RCPF instruction (16-bit reciprocal).

The runtime-support arithmetic functions can use expression registers without saving them. Each function has individual register-use conventions, and the compiler respects these conventions. Register use is fully documented in the source to each of the functions.

The arithmetic functions are written in assembly language. Because of the special calling conventions these functions use, *you cannot access them from C.*

Object code for them is provided in the object library rts30.lib or rts40.lib. Any of these functions that your program needs are linked in automatically if you name one of these libraries as input at link time.

The source code for these functions is in the source library rts.src. The source code has comments that describe the operation and timing of the functions. You can extract, inspect, and modify any of the math functions; be sure you follow the special calling conventions and register-saving rules outlined in this section. Version 30 and version 40 functions may be the same functions.

Table 4–4 summarizes the runtime-support functions used for arithmetic.

Table 4–4. Summary of Runtime-Support Arithmetic Functions

Function	Description	Defined In	Registers Used
MPY_I30	Integer multiply	mpyi.asm	R0, R1, AR0, AR1
DIV_I30	Integer divide	divi.asm	RC, RS, RE
DIV_I40	Integer divide	divi.asm	RC, RS, RE
DIV_U30	Unsigned integer divide	divu.asm	R0, R1, AR0, AR1, RC, RS, RE
DIV_U40	Unsigned integer divide	divu.asm	R0, R1, AR0, AR1, RC, RS,RE
DIV_F30	Floating-point divide	divf.asm	R0, R1, AR0, AR1
DIV_F40	Floating-point divide	divf.asm	R0, R1, AR0, AR1, R10
INV_F30	Floating-point reciprocal	invf.asm	R0, R1, AR0, AR1
INV_F40	Floating-point reciprocal	invf.asm	R0, R1, AR0, AR1
MOD_I30	Integer modulo	modi.asm	R0, R1, AR0, AR1
MOD_I40	Integer modulo	modi.asm	R0, R1, AR0, AR1
MOD_U30	Unsigned integer modulo	modu.asm	R0, R1, AR0, AR1, RC, RS, RE
MOD_U40	Unsigned integer modulo	modu.asm	R0, R1, AR0, AR1, RC, RS,RE
Functions Provided for Compatibility With Previous Compiler Versions			
MPY_I	Integer multiply	arith410.asm	R0–R3
DIV_I	Integer divide	arith410.asm	R0–R3, RC, RS, RE
DIV_U	Unsigned integer divide	arith410.asm	R0–R3, RC, RS, RE
DIV_F	Floating-point divide	arith410.asm	R0–R3
MOD_I	Integer modulo	arith410.asm	R0–R3, RC, RS, RE
MOD_U	Unsigned integer modulo	arith410.asm	R0–R3, RC, RS, RE

4.8 System Initialization

Before you can run a C program, the C runtime environment must be created. This task is performed by the C boot routine, which is a function called c_int00.

The c_int00 function can be branched to, called, or vectored by reset hardware to begin running the system. The function is in the runtime-support library and must be linked with the other C object modules. This occurs automatically when you use the –c or –cr option in the linker and include a runtime-support library created from rts.src as one of the linker input files. When C programs are linked, the linker sets the entry point value in the executable output module to the symbol c_int00.

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

- Defines a section called .stack for the system stack and sets up the initial stack and frame pointers
- Autoinitializes global variables by copying the data from the initialization tables in .cinit to the storage allocated for the variables in .bss. In the small model, the constant tables are also copied from .cinit to .bss. Note this does not refer to data in the .const section

  In the RAM initialization model, a loader performs this step before the program runs (it is not performed by the boot routine). For more information, refer to subsection 4.8.1.
- *Small memory model only* — sets up the page pointer DP to point to the global storage page in .bss
- Calls the function main to begin running the C program

You can replace or modify the boot routine to meet your system requirements. However, the boot routine *must* perform the four operations listed above in order to correctly initialize the C environment. The runtime-support source library contains the source to this routine in a module named boot.asm.

4.8.1 AutoInitialization of Variables and Constants

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

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

Note: Initializing Variables

In standard C, global and static variables that are not explicitly initialized are set to 0 before program execution. The TMS320 floating-point C compiler does not perform any preinitialization of uninitialized variables. Any variable that must have an initial value of 0 must be explicitly initialized. An alternative is to have a loader or boot.obj clear the .bss section before the program starts running.

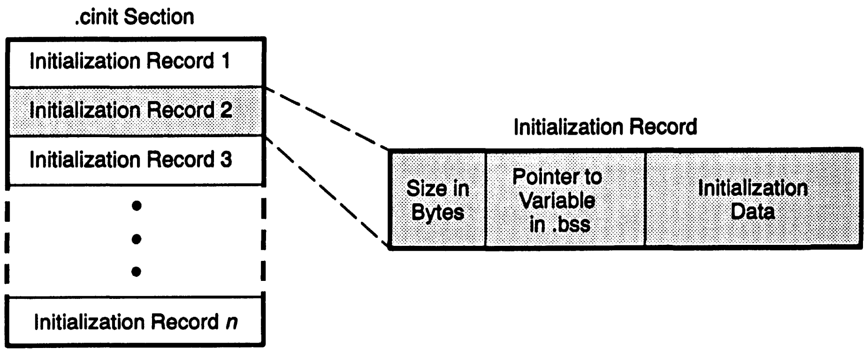
In the small memory model, any tables of long constant values or constant addresses must also be copied into the global data page at this time. Data for these tables is incorporated into the initialization tables in .cinit and thus is automatically copied at initialization time.

There are two methods for copying the initialization data into memory: RAM and ROM. The RAM model of initialization is discussed on page 4-35, and page 4-36 describes the ROM model of initialization.

Initialization Tables

The tables in the .cinit section consist of initialization records with varying sizes. Each initialization record has the following format:

Figure 4–3. Format of Initialization Records in the .cinit Section



- The first field (word 0, record 1) is the size (in words) of the initialization data for the variable.
- The second field (word 1, record 2) is the starting address of the area in the .bss section into which the initialization data must be copied. (This field points to a variable's space in .bss.)
- The third field (word 2, record 3, through *n*) contains the data that is copied into the variable to initialize it.

The .cinit section contains an initialization record for each variable that must be initialized. For example, suppose two initialized variables are defined in C as follows:

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

The initialization tables would appear as follows:

```
      .sect     ".cinit"      ; Initialization section
* Initialization record for variable i
      .word     1             ; Length of data (1 word)
      .word     _i            ; Address in .bss
      .word     23            ; Data to initialize i
* Initialization record for variable a
      .word     5             ; Length of data (5 words)
      .word     _a            ; Address in .bss
      .word     1,2,3,4,5     ; Data to initialize a
```

The .cinit section contains *only* initialization tables in this format. If you are interfacing assembly language modulesto your C program, do not use the .cinit section for any other puporse.

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

Initializing Variables in the RAM Model

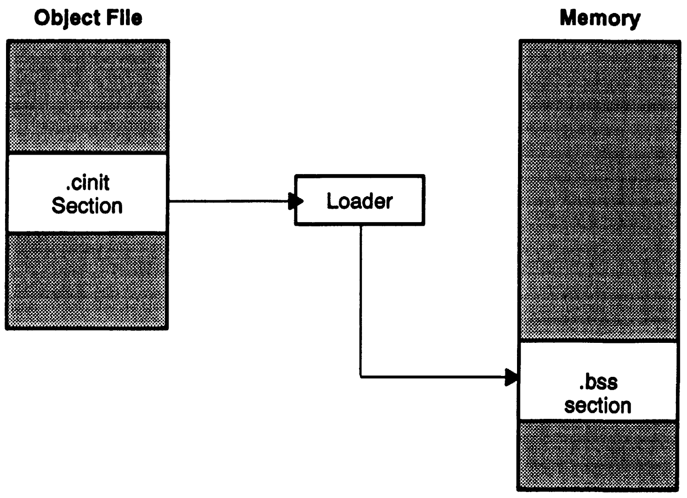
The RAM model, specified with the –cr linker option, allows variables to be initialized at *load time* instead of at runtime. This can enhance performance by reducing boot time and by saving the memory used by the initialization tables. The RAM option requires the use of a **smart loader** to perform the initialization as it copies the program from the object file into memory.

In the RAM model, the linker marks the .cinit section with a special attribute (STYP_CPY equals 1). This means that the section is *not* loaded into memory and does *not* occupy space in the memory map. The linker also sets the symbol cinit to –1 to indicate to the C boot routine that the initialization tables are not present in memory; accordingly, no runtime initialization is performed at boot time.

Instead, when the program is loaded into memory, the loader must detect the presence of the .cinit section and its special attribute. Instead of loading the section into memory, the loader uses the initialization tables directly from the object file to initialize the variables in .bss. To use the RAM model, the loader must understand the format of the initialization tables so that it can use them.

A loader is *not* part of the compiler package. The loader used in TI emulator and simulator products is a smart loader.

Figure 4–4. RAM Model of Autoinitialization

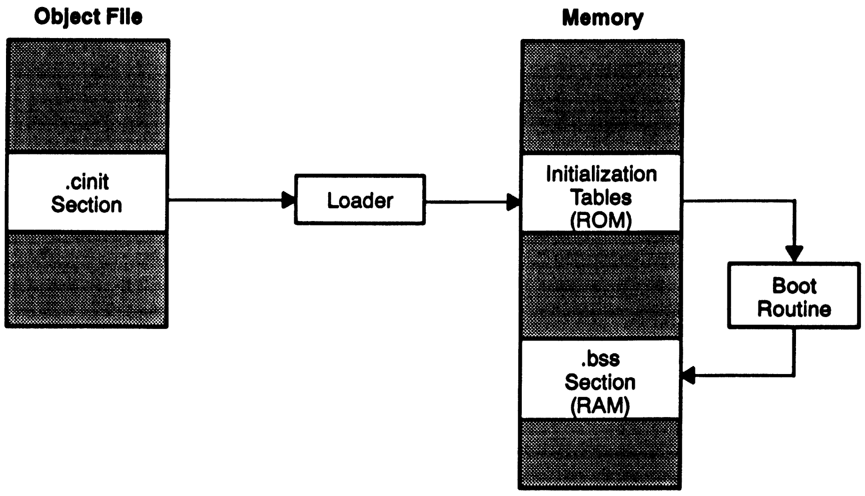


Initializing Variables in the ROM Model

The ROM model is the default model for autoinitialization. To use the ROM model, invoke the linker with the –c option.

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

Figure 4–5. ROM Model of Autoinitialization



Chapter 5

Runtime-Support Functions

Some of the tasks that a C program must perform (such as I/O, floating-point arithmetic, dynamic memory allocation, string operations, and trigonometric functions) are not part of the C language itself. The runtime-support functions, which are included with the C compiler, are standard ANSI functions that perform these tasks. The runtime-support library, rts.src, contains the source for these functions, as well as for other functions and routines. If you use any of the runtime-support functions, be sure to build the appropriate library, according to the desired runsime model, using the library-build utility; then include that library as linker input when you link your C program.

These are the topics covered in this chapter:

Topic **Page**

5.1 Libraries

Three libraries are included with the TMS320 floating-point C compiler: two object libraries containing object code for the runtime-support and a source library containing source code for the functions in the object libraries.

- **rts30.lib** is the standard runtime-support object library for the TMS320C3x. It was built from the source library with the –v30, –o2, –x, and –mi options. You can link this library with any files compiled for the 'C3x with the small memory model and the standard runtime model.
- **rts40.lib** is the standard runtime-support object library for the TMS320C4x. It was built from the source library with the –v40, –o2, –x, and –mi options. You can link this library with any files compiled for the 'C4x with the small memory model and the standard runtime model.
- **rts.src** is the source library. If you use code compiled with the large runtime model or with the register-argument runtime model (or both), you must create your own runtime-support library from rts.src by using the library-build utility (mk30) described in Chapter 6.

All object libraries built from rts.src include the standard C runtime-support functions described in this chapter, the intrinsic arithmetic routines described in Section 4.7 on page 4-31, and the system startup routine, _c_int00. The object libraries are built from the C and assembly source contained in rts.src.

When you link your program, you must specify an object library as one of the linker input files so that references to runtime-support functions can be resolved. You should usually specify libraries *last* on the linker command line because the assembler searches for unresolved references when it encounters a library on the command line. You can also use the –x linker option to force repeated searches of each library until it 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, refer to Section 2.9 on page 2-54.

5.1.1 Modifying a Library Function

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

```
ar30 x rts.src atoi.c strcpy.c
```

To modify a function, extract the source as in the previous example. Make the required changes to the code, recompile, and then reinstall the new object file or files into the library:

```
cl30 <–options> atoi.c strcpy.c          ;recompile
ar30 r rts.lib  atoi.obj strcpy.obj      ;rebuild library
```

You can also build a new library this way, rather than rebuilding back into rts.lib. For more information about the archiver, refer to Chapter 8 of the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide.*

5.1.2 Building a Library With Different Options

You can create a new library from rts.src by using the library-build utility, mk30. For example, use this command to build an optimized runtime-support library for the TMS320C4x using the big memory model and the register-argument runtime model:

```
mk30 ––u –mr –v40 –mb –o2 –x rts.src –l rts40rb.lib
```

The –u option tells the mk30 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 'C4x (–v40) using the big memory model (–mb) and the register-argument runtime model (–mr). The use of the optimizer (–o2) and inline function expansion (–x) options does not affect compatibility with code compiled without these options.

5.2 Header Files

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

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

These are the header files that declare the runtime-support functions:

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

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

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

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

Subsections 5.2.1 through 5.2.11 describe the header files that are included with the TMS320 floating-point C compiler. Section 5.3, page 5-12, lists the functions that these headers declare.

5.2.1 Diagnostic Messages (assert.h)

The *assert.h* header defines the assert macro, which inserts diagnostic failure messages into programs at runtime. The assert macro tests a runtime expression.

- ❑ If the expression is true (nonzero), the program continues running.
- ❑ If the expression is false, the macro outputs a message that contains the expression, the source file name, and the line number of the statement that contains the expression; then, the program terminates (via the abort function).

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

5.2.2 Character-Typing and Conversion (ctype.h)

The *ctype.h* header declares functions that test and convert characters.

For example, a character-typing function may test a character to determine whether it is a letter, a printing character, a hexadecimal digit, etc. These functions return a value of *true* (a nonzero value) or *false* (0).

The character-conversion functions convert characters to lower case, upper case, or ASCII and return the converted character.

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

The ctype.h header also contains macro definitions that perform these same operations; the macros run faster than the corresponding functions. 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*).

5.2.3 Limits (float.h and limits.h)

The *float.h* and *limits.h* headers define macros that expand to useful limits and parameters of the TMS320C3x/C4x's numeric representations. Table 5–1 and Table 5–2 list these macros and the limits with which they are associated.

Table 5–1. Macros That Supply Integer Type Range Limits (limits.h)

Macro	Value	Description
CHAR_BIT	32	Number of bits in type char
SCHAR_MIN	–2147483648	Minimum value for a signed char
SCHAR_MAX	2147483647	Maximum value for a signed char
UCHAR_MAX	4294967295	Maximum value for an unsigned char
CHAR_MIN	SCHAR_MIN	Minimum value for a char
CHAR_MAX	SCHAR_MAX	Maximum value for a char
SHRT_MIN	–2147483648	Minimum value for a short int
SHRT_MAX	2147483647	Maximum value for a short int
USHRT_MAX	4294967295	Maximum value for an unsigned short int
INT_MIN	–2147483648	Minimum value for an int
INT_MAX	2147483647	Maximum value for an int
UINT_MAX	4294967295	Maximum value for an unsigned int
LONG_MIN	–2147483648	Minimum value for a long int
LONG_MAX	2147483647	Maximum value for a long int
ULONG_MAX	4294967295	Maximum value for an unsigned long int

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

Table 5–2. Macros That Supply Floating-Point Range Limits (float.h)

Macro	Value	Description
FLT_RADIX	2	Base or radix of exponent representation
FLT_ROUNDS	–1	Rounding mode for floating-point addition
FLT_DIG DBL_DIG LDBL_DIG	6	Number of decimal digits of precision for a float, double, or long double
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	24	Number of base-FLT_RADIX digits in the mantissa of a float, double, or long double
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	–126	Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized float, double, or long double
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	128	Maximum negative integer such that FLT_RADIX raised to that power minus 1 is a representable finite float, double, or long double
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	$1.1920929E^{-07}$	Minimum positive float, double, or long double number x such that $1.0 + x \neq 1.0$
FLT_MIN DBL_MIN LDBL_MIN	$5.8774817E^{-39}$	Minimum positive float, double, or long double
FLT_MAX DBL_MAX LDBL_MAX	$3.4028235E^{+38}$	Maximum float, double, or long double
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	–39	Minimum negative integers such that 10 raised to that power is in the range of normalized floats, doubles, or long doubles
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	38	Maximum positive integers such that 10 raised to that power is in the range of representable finite floats, doubles, or long doubles

Key to prefixes:

FLT_	applies to type float
DBL_	applies to type double
LDBL_	applies to type long double

5.2.4 Floating-Point Math (math.h)

The *math.h* header defines several trigonometric, exponential, and hyperbolic math functions. These math functions expect double precision floating point arguments and return double precision floating point values. Except where indicated, all trigonometric functions use angles expressed in radians.

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

5.2.5 Error Reporting (errno.h)

Errors can occur in a math function if the 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* and defined in *errno.c.*

5.2.6 Variable Arguments (stdarg.h)

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

- The three macros are va_start, va_arg, and va_end. These macros are used when the number and type of arguments may vary each time a function is called.

- The type, *va_list*, is a pointer type that can hold information for va_start, va_end, and va_arg.

A variable-argument function can use the macros declared by stdarg.h to step through its argument list at runtime, when it knows the number and types of arguments actually passed to it.

5.2.7 Standard Definitions (stddef.h)

The *stddef.h* header defines two types and two macros. The types include:

- *ptrdiff_t*, a signed integer type that is the resul from the subtraction of two pointers
- *size_t*, an unsigned integer type that is the data type of the *sizeof* operator

The macros include:

- The NULL macro, which expands to a null pointer constant(0)
- The *offsetof(type, identifier)* macro, which expands to an integer that has type *size_t*. The result is the value of an offset in bytes to a structure member (identifier) from the beginning of its structure (type)

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

5.2.8 General Utilities (stdlib.h)

The *stdlib.h* header declares several functions, one macro, and two types. The macro is named RAND_MAX. The types include:

- *div_t*, a structure type that is the type of the value returned by the div function
- *ldiv_t*, a structure type that is the type of the value returned by the ldiv function

The stdlib.h header also declares many of the common library functions:

- **String conversion** functions that convert strings to numeric representations
- **Searching and sorting** functions that allow you to search and sort arrays
- **Sequence-generation** functions that allow you to generate a pseudorandom sequence and allow you to choose a starting point for a sequence
- **Program-exit functions** that allow your program to terminate normally or abnormally
- **Integer-arithmetic** that is not provided as a standard part of the C language

5.2.9 String Functions (string.h)

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

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

In C, all character strings are terminated with a 0 (null) character. The string functions named **str***xxx* all operate according to this convention. Additional functions that are also declared in string.h allow you to perform corresponding operations on arbitrary sequences of bytes (data objects), where a 0 value does not terminate the object. These functions have names such as **mem***xxx*.

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

5.2.10 Time Functions (time.h)

The *time* header declares one macro, several types, and functions that manipulate dates and times. Times are represented in two different 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 is a synonym for the type unsigned long.

- As a structure of type *struct_tm*. This structure contains members for expressing time as a combination of years, months, days, hours, minutes, and seconds. A time represented like this is called broken-down time. The structure has the following members.

```
int    tm_sec;      /* seconds after the minute (0–59) */
int    tm_min;      /* minutes after the hour (0–59)   */
int    tm_hour;     /* hours after midnight (0–23)     */
int    tm_mday;     /* day of the month (1–31)         */
int    tm_mon;      /* months since January (0–11)     */
int    tm_year;     /* years since 1900 (0–99)         */
int    tm_wday;     /* days since Saturday (0–6)       */
int    tm_yday;     /* days since January 1 (0–365)    */
int    tm_isdst;    /* Daylight Savings Time flag      */
```

A time, whether represented as a time_t or a struct tm, can be expressed from different points of reference.

- Calendar time represents the current Gregorian date and time.

- Local time is the calendar time expressed for a specific time zone.

Local time may be adjusted for daylight savings time. Obviously, local time depends on the time zone. The time.h header declares a structure type called tmzone and a variable of this type called _tz. You can change the time zone by modifying this structure, either at runtime or by editing tmzone.c and changing the initialization. The default time zone is U.S. central standard time.

The basis for all the functions in time.h are two system functions: clock and time. Time provides the current time (in time_t format), and clock provides the system time (in arbitrary units). The value returned by clock can be divided by the macro CLOCKS_PER_SEC to convert it to seconds. Since these functions and the CLOCKS_PER_SEC macro are system specific, only stubs are provided in the library. To use the other time functions, you must supply custom versions of these functions.

5.2.11 Nonlocal Jumps (setjmp.h)

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

- *jmpbuf*, an array type suitable for holding the information needed to restore a calling environment.
- *setjmp*, a macro that saves its calling environment in its jmp_buf argument for later use by the longjmp function.
- *longjmp*, a function that uses its jmp_buf argument to restore the program environment.

5.3 Summary of Runtime-Support Functions and Macros

Refer to the following pages for information about functions and macros:

Error Message Macro (assert.h)	**Description**
void **assert**(int expression);‡	Inserts diagnostic messages into programs

Character-Typing Conversion Functions (ctype.h)		**Description**
int	**isalnum**(char c);†	Tests c to see if it's an alphanumeric ASCII character
int	**isalpha**(char c);†	Tests c to see if it's an alphabetic ASCII character
int	**isascii**(char c);†	Tests c to see if it's an ASCII character
int	**iscntrl**(char c);†	Tests c to see if it's a control character
int	**isdigit**(char c);†	Tests c to see if it's a numeric character
int	**isgraph**(char c); †	Tests c to see if it's any printing character except a space
int	**islower**(char c); †	Tests c to see if it's a lowercase alphabetic ASCII character
int	**isprint**(char c);†	Tests c to see if it's a printable ASCII character (including spaces)
int	**ispunct**(char c);†	Tests c to see if it's an ASCII punctuation character
int	**isspace**(char c);†	Tests c to see if it's an ASCII spacebar, tab (horizontal or vertical), carriage return, formfeed, or newline character
int	**isupper**(char c);†	Tests c to see if it's an uppercase ASCII alphabetic character
int	**isxdigit**(char c);†	Tests c to see if it's a hexadecimal digit
char	**toascii**(char c);†	Masks c into a legal ASCII value
char	**tolower**(int char c);†	Converts c to lowercase if it's uppercase
char	**toupper**(int char c);†	Converts c to uppercase if it's lowercase

Floating-Point Math Functions (math.h)		**Description**
double	**acos**(double x);	Returns the arccosine of x
double	**asin**(double x);	Returns the arcsine of x
double	**atan**(double x);	Returns the arctangent of x
double	**atan2**(double y, double x);	Returns the arctangent of y/x
double	**ceil**(double x);†	Returns the smallest integer greater than or equal to x
double	**cos**(double x);	Returns the cosine of x
double	**cosh**(double x);	Returns the hyperbolic cosine of x
double	**exp**(double x);	Returns the exponential function of x
double	**fabs**(double x);§	Returns the absolute value of x
double	**floor**(double x);†	Returns the largest integer less than or equal to x
double	**fmod**(double x, double y);†	Returns the floating-point remainder of x/y

† Expands inline if –x is used
‡ Macro
§ Expands inline unless –x0 is used

Floating-Point Math Functions (continued)		**Description**
double	**frexp**(double value, int *exp);	Breaks value into a normalized fraction and an integer power of 2
double	**ldexp**(double x, int exp);	Multiplies x by an integer power of 2
double	**log**(double x);	Returns the natural logarithm of x
double	**log10**(double x);	Returns the base-10 logarithm of x
double	**modf**(double value, int *iptr);	Breaks value into a signed integer and a signed fraction
double	**pow**(double x, double y);	Returns x raised to the power y
double	**sin**(double x);	Returns the sine of x
double	**sinh**(double x);	Returns the hyperbolic sine of x
double	**sqrt**(double x);	Returns the nonnegative square root of x
double	**tan**(double x);	Returns the tangent of x
double	**tanh**(double x);	Returns the hyperbolic tangent of x

Variable-Argument Functions and Macros (stdarg.h)		**Description**
type	**va_arg**(va_list ap);‡	Accesses the next argument of type *type* in a variable-argument list
void	**va_end**(va_list ap);‡	Resets the calling mechanism after using va_arg
void	**va_start**(va_list ap);‡	Initializes ap to point to the first operand in the variable-argument list

General Utilities(stdlib.h)		**Description**
int	**abs**(int j);§	Returns the absolute value of j
void	**abort**(void)	Terminates a program abnormally
void	**atexit**(void (*fun)(void));	Registers the function pointed to by fun, to be called without arguments at normal program termination
double	**atof**(char *nptr);	Converts a string to a floating-point value
int	**atoi**(char *nptr);	Converts a string to an integer value
long	**atol**(char *nptr);	Converts a string to a long integer value
void	***bsearch**(void *key, void *base, size_t n, size_t size, int (*compar) (void));	Searches through an array of n objects for the object that key points to
void	***calloc** (size_t n, size_t size);	Allocates and clears memory for n objects, each of size bytes
div_t	**div**(int numer, int denom);	Divides numer by denom producing a quotient and a remainder
void	**exit**(int status);	Terminates a program normally
void	**free**(void *ptr);	Deallocates memory space allocated by malloc, calloc, or realloc

† Expands inline if –x is used
‡ Macro
§ Expands inline unless –x0 is used

General Utilities (continued)		Description
long	**labs**(long j);§	Returns the absolute value of j
ldiv_t	**ldiv** (long numer, long denom);	Divides numer by denom producing a quotient and a remainder
int	**ltoa**(long n, char *buffer);	Converts n to the equivalent digit string
void	***malloc**(size_t size);	Allocates memory for an object of size bytes
void	**minit**(void);	Resets all the memory previously allocated by malloc, calloc, or realloc
void	**qsort**(void *base, size_t n, size_t _size, int (*_compar) (void));	Sorts an array of n members; base points to the first member of the unsorted array, and size specifies the size of members
int	**rand**(void);	Returns a sequence of pseudorandom integers in the range 0 to RAND_MAX
void	***realloc**(void *ptr, size_t size);	Changes the size of memory pointed to by ptr to size bytes
void	**srand**(unsigned seed);	Resets the random number generator
double	**strtod**(char *nptr, char **endptr);	Converts a string to a floating-point value
long	**strtol**(char *nptr, char **endptr, int base);	Converts a string to a long integer value
unsigned long	**strtoul**(char *nptr, char **endptr, int base);	Converts a string to an unsigned long integer value

String Functions(string.h)		Description
void	***memchr**(void *s, int c, size_t n);†	Finds the first occurrence of c in the first n characters of s
int	**memcmp**(void *s1,void *s2, size_t n);†	Compares the first n characters of s1 to s2
void	***memcpy**(void *s1, void *s2, size_t n);†	Copies n characters from s1 to s2
void	***memmove**(void *s1, void *s2, size_t n);	Moves n characters from s1 to s2
void	***memset**(void *s, int c, size_t n);†	Copies the value of c into the first n characters of s
char	***strcat**(char *s1, char *s2);†	Appends s2 to the end of s1
char	***strchr**(char *s, int c);†	Finds the first occurrence of character c in s
int	**strcmp**(char *s1, char *s2);†	Compares strings and returns one of the following values: <0 if s1 is less than s2; =0 if s1 is equal to s2; >0 if s1 is > s2
int	***strcoll**(char *s1, char *s2);	Compares strings and returns one of the following values, depending on the locale: <0 if s1 is less than s2; =0 if s1 is equal to s2; >0 if s1 is greater than s2
char	***strcpy**(char *s1, char *s2);†	Copies string s2 into s1

† Expands inline if –x is used
‡ Macro
§ Expands inline unless –x0 is used

String Functions (continued)		Description
size_t	**strcspn**(char *s1, char *s2);	Returns the length of the initial segment of s1 that is made up entirely of characters that are not in s2
char	***strerror**(int errnum);	Maps the error number in errnum to an error message string
size_t	**strlen**(char *s);†	Returns the length of a string
char	***strncat**(char *s1, char *s2, size_t n);	Appends up to n characters from s1 to s2
int	**strncmp**(char *s1, char *s2, size_t n);	Compares up to n characters in two strings
char	***strncpy**(char *s1, char *s2, size_t n);	Copies up to n characters of a s2 to s1
char	***strpbrk**(char *s1, char *s2);	Locates the first occurrence in s1 of *any* character from s2
char	***strrchr**(char *s, char c);†	Finds the last occurrence of character c in s
size_t	**strspn**(char *s1, char *s2);	Returns the length of the initial segment of s1, which is entirely made up of characters from s2
char	***strstr**(char *s1, char *s2);	Finds the first occurrence of s2 in s1
char	***strtok**(char *s1, char *s2);	Breaks s1 into a series of tokens, each delimited by a character from s2
Nonlocal Jumps (setjmp.h)		**Description**
int	**setjmp**(jmp_buf env);‡	Saves calling environment for later use by longjmp function
void	**longjmp**(jmp_buf env, int returnval);	Uses jmp_buf argument to restore a previously saved environment
Time Functions (time.h)		**Description**
char	***asctime**(struct tm *timeptr);	Converts a time to a string
clock_t	**clock**(void);	Determines the processor time used
char	***ctime**(struct time *timeptr);	Converts time to a string
double	**difftime**(time_t time1, time_t time0);	Returns the difference between two calendar times
struct tm	***gmtime**(time_t *timer);	Converts local time to Greenwich Mean Time
struct tm	***localtime**(time_t *timer);	Converts time_t value to broken down time
time_t	**mktime**(struct tm *timeptr);	Converts broken down time to a time_t value
size_t	**strftime**(char *s,*format, size_t maxsize, struct tm *timeptr);	Formats a time into a character string
time_t	**time**(time_t *timer);	Returns the current calendar time

† Expands inline if –x is used
‡ Macro
§ Expands inline unless –x0 is used

5.4 Functions Reference

The remainder of this chapter is a reference.

Functions	Page	Functions	Page
abort	5-18	ltoa	5-34
abs	5-18	malloc	5-35
acos	5-19	memchr	5-35
asctime	5-19	memcmp	5-36
asin	5-19	memcpy	5-36
assert	5-20	memmove	5-36
atan	5-21	memset	5-37
atan2	5-21	minit	5-37
atexit	5-21	mktime	5-39
atof	5-22	modf	5-39
atoi	5-22	pow	5-39
atol	5-22	qsort	5-40
bsearch	5-23	rand	5-41
calloc	5-24	realloc	5-41
ceil	5-24	setjmp	5-42
clock	5-24	sin	5-43
cos	5-25	sinh	5-43
cosh	5-26	sqrt	5-43
ctime	5-26	srand	5-41
difftime	5-26	strcat	5-44
div	5-27	strchr	5-44
exit	5-28	strcmp	5-45
exp	5-28	strcoll	5-45
fabs	5-28	strcpy	5-46
floor	5-29	strcspn	5-46
fmod	5-29	strerror	5-47
free	5-30	strftime	5-47
frexp	5-30	strlen	5-49
gmtime	5-30	strncat	5-50
isalnum	5-32	strncmp	5-51
isalpha	5-32	strncpy	5-51
isascii	5-32	strpbrk	5-53
iscntrl	5-32	strrchr	5-53
isdigit	5-32	strspn	5-54
isgraph	5-32	strstr	5-54
islower	5-32	strtod	5-55
isprint	5-32	strtok	5-56
ispunct	5-32	strtol	5-55
isspace	5-32	strtoul	5-55
isupper	5-32	tan	5-56
isxdigit	5-32	tanh	5-57
labs	5-18	time	5-57
ldexp	5-33	toascii	5-58
ldiv	5-27	tolower	5-58
localtime	5-27	toupper	5-58
log	5-33	va_arg	5-59
log10	5-34	va_end	5-59
longjmp	5-42	va_start	5-59

abort

Abort

Syntax

#include <stdlib.h>

void abort(void);

Defined in

exit.c in rts.src

Description

The **abort** function usually terminates a program with an error code. The TMS320C3x/C4x implementation of the abort function calls the exit function with a value of 0, and is defined as follows:

```
void abort ()
{
   exit(0);
}
```

This makes the abort function equivalent to the exit function.

abs/labs

Absolute Value

Syntax

#include <stdlib.h>

int abs(int j);
long int labs(long int k);

Defined in

abs.c in rts.src

Description

The C compiler supports two functions that return the absolute value of an integer:

- The **abs** function returns the absolute value of an integer j.
- The **labs** function returns the absolute value of a long integer k.

Since int and long int are functionally equivalent types in TMS320 floating-point C, the abs and labs functions are also functionally equivalent. The abs and labs functions are expanded inline unless the –x0 option is used.

Example

```
int x = -5;
int y = abs (x);     /* abs returns 5 */
```

acos

Arc Cosine

Syntax

#include <math.h>

double acos(double x);

Defined in

acos.c in rts.src

Description

The **acos** function returns the arc cosine of a floating-point argument x. x must be in the range [–1,1]. The return value is an angle in the range [0,π] radians.

Example

```
double realval, radians;

realval = 0.0;
radians = acos(realval); /* acos return π/2 */
return (radians);
```

asctime

Internal Time to String

Syntax

#include <time.h>

char *asctime(struct tm *timeptr);

Defined in

asctime.c in rts.src

Description

The **asctime** function converts a broken-down time into a string with the following form:

```
Mon Jan 11 11:18:36 1988 \n\0
```

The function returns a pointer to the converted string.

For more information about the functions and types that the time.h header declares, refer to subsection 5.2.10, page 5-10.

asin

Arc Sine

Syntax

#include <math.h>

double asin(double x);

Defined in

asin.c in rts.src

Description

The **asin** function returns the arc sine of a floating-point argument x. x must be in the range [–1,1]. The return value is an angle in the range [–π/2,π/2] radians.

Example

```
double realval, radians;

realval = 1.0;
radians = asin(realval);  /* asin returns π/2 */
```

assert — *Insert Diagnostic Information Macro*

Syntax

#include <assert.h>

void assert(int expression);

Defined in

assert.h as a macro

Description

The **assert** macro tests an expression; depending upon the value of the expression, assert either aborts execution and issues a message or continues execution. This macro is useful for debugging.

- If expression is false, the assert macro writes information about the particular call that failed to the standard output, and then aborts execution.
- If expression 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, then the assert macro is defined to have no effect.

If NDEBUG is not defined when assert.h is included, the assert macro is defined to test the expression and, if false, write a diagnostic message including the source filename, line number, and test of expression.

The assert macro is defined with the printf function, which is not included in the library. To use assert, you must either:

- Provide your own version of printf, or
- Modify assert to output the message by other means

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 when this code runs, a message such as:

```
Assertion failed (j), file foo.c, line 123
```

is sent to standard output.

```
int   i, j;
assert(j);
q = i/j;
```

atan — *Polar Arc Tangent*

Syntax

#include <math.h>

double atan(double x);

Defined in

atan.c in rts.src

Description

The **atan** function returns the arc tangent of a floating-point argument x. The return value is an angle in the range [−π/2,π/2] radians.

Example

```
double realval, radians;

realval = 1.0;
radians = atan(realval);     /* return value =  π/4*/
```

atan2 — *Cartesian Arc Tangent*

Syntax

#include <math.h>

double atan2(double y, x);

Defined in

atan.c in rts.src

Description

The **atan2** function returns the arc tangent of y/x. The function uses the signs of the arguments to determine the quadrant of the return value. Both arguments cannot be 0. The return value is an angle in the range [−π,π] radians.

Example

```
atan2 (1.0, 1.0)       /* returns π/4 */
atan2 (1.0, −1.0)      /* returns 3π/4 */
atan2 (−1.0, 1.0)      /* returns ±π/4 */
atan2 (−1.0, −1.0)     /* returns −3π/4 */
```

atexit — *Exit Without Arguments)*

Syntax

#include <stdlib.h>

void 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, a call to abort, or a return from the main function, the functions that were registered are called, without arguments, in reverse order of their registration.

atof/atoi/atol — *Convert ASCII to Number*

Syntax

#include <stdlib.h>

double atof(char *nptr);
int atoi(char *nptr);
long int atol(char *nptr);

Defined in

atof.c and atoi.c in rts.src

Description

Three functions convert strings to numeric representations:

- ❑ The **atof** function converts a string into a floating-point value. Argument nptr 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 nptr 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 nptr points to the string; the string must have the following format:

  [space] [sign] digits

The *space* is indicated by one or more of the following characters: a space (character), a horizontal or vertical tab, a carriage return, a form feed, or a newline. Following the *space* is an optional *sign*, and then *digits* that represent the integer portion of the number. The fractional part of the number follows, then the exponent, including an optional *sign*.

The first character that cannot be part of the number terminates the string.

Since int and long are functionally equivalent in TMS320 floating-point C, the atoi and atol functions are also functionally equivalent.

The functions do not handle any overflow resulting from the conversion.

Example

```
int i;
double d;
i = atoi ("-3291");       /* i = -3291 */
d = atof ("1.23e-2);      /* d = .0123 */
```

bsearch — *Array Search*

Syntax

#include <stdlib.h>

void *bsearch(void *key, void *base, size_t nmemb,
size_t size, int (*compar)(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, sorted 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 user-defined function that compares key to the array elements. The comparison function should be declared as:

```
int  cmp(const void *ptr1, const void *ptr2);
```

The cmp function compares the objects that prt1 and ptr2 point to and returns one of the following values:

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

In order for the search to work properly, the comparison must return <0, 0, >0.

Example

```
#include <stdlib.h>
#include <stdio.h>

int list [] = {1, 3, 4, 6, 8, 9};
int diff (const void *, const void *0;

main()
{
    int key = 8;
    int p = bsearch (&key, list, 6, 1, idiff);
        /* p points to list[4] */
}
int idiff (const void *i1, const void *i2)
{
    return *(int *) i1 - *(int *) i2;
}
```

calloc — *Allocate and Clear Memory*

Syntax

#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);

Defined in

memory.c in rts.src

Description

The **calloc** function allocates size bytes for each of nmemb 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, defined in an uninitialized named section called .sysmem in memory.c. The linker sets the size of this section from the value specified by the –heap option. Default heap size is 1 K words. For more information, refer to subsection 4.1.3, *Dynamic Memory Allocation*, on page 4-4.

Example

This example uses the calloc routine to allocate and clear 10 bytes.

```
ptr = calloc (10,2);    /*Allocate and clear 20 bytes */
```

ceil — *Ceiling*

Syntax

#include <math.h>

double ceil(double x);

Defined in

ceil.c in rts.src

Description

The **ceil** function returns a floating-point number that represents the smallest integer greater than or equal to x. The ceil function is inlined if the –x2 option is used.

Example

```
double answer;
answer = ceil(3.1415);   /* answer = 4.0 */
answer = ceil(-3.5);     /* answer = -3.0 */
```

clock

Processor Time

Syntax

#include <time.h>

clock_t clock(void);

Defined in

clock.c in rts.src

Description

The **clock** function determines the amount of processor time used. It returns an approximation of the processor time used by a program since the program began running. The return value can be converted to seconds by dividing 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 –1.

Note: Writing Your Own Clock Function

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

For more information about the functions and types that the time.h header declares, refer to subsection 5.2.10, page 5-10.

cos

Cosine

Syntax

#include <math.h>

double cos(double x);

Defined in

cos.c in rts.src

Description

The **cos** function returns the cosine of a floating-point number x. The angle x is expressed in radians. An argument with a large magnitude may produce a result with little or no significance.

Example

```
double radians, cval;      /* cos returns cval */
radians = 3.1415927;
cval = cos(radians);       /* return value = -1.0 */
```

cosh — *Hyperbolic Cosine*

Syntax

#include <math.h>

double cosh(double x);

Defined in

cosh.c in rts.src

Description

The **cosh** function returns the hyperbolic cosine of a floating-point number x. A range error occurs if the magnitude of the argument is too large.

Example

```
double x, y;

x = 0.0;
y = cosh(x);  /* return value = 1.0 */
```

ctime — *Calendar Time*

Syntax

#include <time.h>

char *ctime(time_t *timer);

Defined in

ctime.c in rts.src

Description

The **ctime** function converts the calendar time (pointed to by timer and represented as a value of type time_t) to a string. This is equivalent to:

```
asctime(localtime(timer))
```

The function returns the pointer returned by the asctime function.

For more information about the functions and types that the time.h header declares, refer to subsection 5.2.10, page 5-10.

difftime — *Time Difference*

Syntax

#include <time.h>

double difftime(time_t time1, time_t time0);

Defined in

difftime.c in rts.src

Description

The **difftime** function calculates the difference between two calendar times, time1 minus time0. The return value is expressed in seconds.

For more information about the functions and types that the time.h header declares, refer to subsection 5.2.10, page 5-10.

div/ldiv — *Division*

Syntax

#include <stdlib.h>

div_t div(int numer, long denom);
ldiv_t ldiv(long numer, long denom);

Defined in

div.c in rts.src

Description

Two functions support integer division by returning numer divided by denom. 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;
```

If the division produces a remainder, the sign of the quotient is the same as the algebraic quotient, and the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient. The sign of the remainder is the same as the sign of numer.

Because ints and longs are equivalent types in TMS320 floating-point C, these functions are also equivalent.

Example

```
int i = -10
int j = 3;
div_t result = div (i, j);/* result.quot == -3 */
                          /* result.rem == -1 */
```

exit — *Normal Termination*

Syntax

#include <stdlib.h>

void exit(int status);

Defined In

exit.c in rts.src

Description

The **exit** function terminates a program normally. All functions registered by the atexit function are called in reverse order of their registration.

You can modify the exit function to perform application-specific shutdown tasks. The unmodified function simply runs in an infinite loop until the system is reset.

Note that the exit function cannot return to its caller.

exp — *Exponential*

Syntax

#include <math.h>

double exp(double x);

Defined In

exp.c in rts.src

Description

The **exp** function returns the exponential function of 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 = 7.38905, which is e**2 */
```

fabs — *Absolute Value*

Syntax

#include <math.h>

double fabs(double x);

Defined In

fabs.c in rts.src

Description

The **fabs** function returns the absolute value of a floating-point number x. The fabs function is expanded inline unless the –x0 option is used.

Example

```
double x, y;

x = –57.5;
y = fabs(x);         /* return value = +57.5 */
```

floor

Floor

Syntax

#include <math.h>

double floor(double x);

Defined in

floor.c in rts.src

Description

The **floor** function returns a floating-point number that represents the largest integer less than or equal to x. The floor function is expanded inline if the –x option is used.

Example

```
double answer;

answer = floor(3.1415);      /* answer = 3.0 */
answer = floor(–3.5);        /* answer = –4.0 */
```

fmod

Floating-Point Remainder

Syntax

#include <math.h>

double fmod(double x, double y);

Defined in

fmod.c in rts.src

Description

The **fmod** function returns the remainder after dividing x by y an integral number of times. 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 */
```

free — *Deallocate Memory*

Syntax

#include <stdlib.h>

void free(void *ptr);

Defined in

memory.c in rts.src

Description

The **free** function deallocates memory space (pointed to by ptr) that was previously allocated by a malloc, calloc, or realloc call. This makes the memory space available again. If you attempt to free unallocated space, the function takes no action and returns. For more information, refer to subsection 4.1.3, *Dynamic Memory Allocation*, on page 4-4.

Example

This example allocates 10 bytes and then frees them.

```
char *x;
x = malloc(10);        /*  allocate 10 bytes    */
free(x);               /*  free 10 bytes        */
```

frexp — *Fraction and Exponent*

Syntax

#include <math.h>

double frexp(double value, int *exp);

Defined in

frexp30.asm in rts.src

Description

The **frexp** function breaks a floating-point number into a normalized fraction and an integer power of 2. The function returns a number x, with a magnitude in the range [1/2,1) or 0, so that value == $x \times 2^{exp}$. The frexp function stores the power in the int pointed to by exp. If value is 0, both parts of the result are 0.

Example

```
double fraction;
int exp;

fraction = frexp(3.0, &exp);

/* after execution, fraction is .75 and exp is 2 */
```

gmtime — *Greenwich Mean Time*

Syntax

#include <time.h>

struct tm *gmtime(time_t *timer);

Defined in

gmtime.c in rts.src

Description

The **gmtime** function converts a local time pointed to by timer into Greenwich Mean Time (represented as a broken-down time).

The adjustment from local time to GMT is dependent on the local time zone. The current time zone is represented by a structure called _tz, of type struct tmzone, defined in tmzone.c. Change this structure for the appropriate time zone.

For more information about the functions and types that the time.h header declares, refer to subsection 5.2.10, page 5-10.

isxxx — *Character Typing*

Syntax

#include <ctype.h>

int isalnum(char c);	**int islower**(char c);
int isalpha(char c);	**int isprint**(char c);
int isascii(char c);	**int ispunct**(char c);
int iscntrl(char c);	**int isspace**(char c);
int isdigit(char c);	**int isupper**(char c);
int isgraph(char c);	**int isxdigit**(char c);

Defined in

isxxx.c and ctype.c in rts.src
Also defined in ctype.h as macros

Description

These functions test a single argument c to see if it is a particular type of character—alphabetic, alphanumeric, numeric, ASCII, etc. If the test is true (the character is the type of character that it was tested to be), the function returns a nonzero value; if the test is false, the function returns 0. All of the character-typing functions are expanded inline if the –x option is used. The character-typing functions include:

isalnum	identifies alphanumeric ASCII characters (tests for any character for which isalpha or isdigit is true).
isalpha	identifies alphabetic ASCII characters (tests for any character for which islower or isupper is true).
isascii	identifies ASCII characters (any character from 0–127).
iscntrl	identifies control characters (ASCII characters 0–31 and 127).
isdigit	identifies numeric characters between 0 and 9 (inclusive).
isgraph	identifies any non-space character.
islower	identifies lowercase alphabetic ASCII characters.
isprint	identifies printable ASCII characters, including spaces (ASCII characters 32–126).
ispunct	identifies ASCII punctuation characters.
isspace	identifies ASCII spacebar, tab (horizontal or vertical), carriage return, form feed, and new line characters.
isupper	identifies uppercase ASCII alphabetic characters.
isxdigit	identifies hexadecimal digits (0–9, a–f, A–F).

The C compiler also supports a set of macros that perform these same functions. The macros have the same names as the functions but are prefixed with an underscore; for example, _isascii is the macro equivalent of the isascii function. In general, the macros execute more efficiently than the functions.

ldexp — *Multiply by a Power of Two*

Syntax

#include <math.h>

double ldexp(double x, int exp);

Defined in

ldexp30.asm in rts.src

Description

The **ldexp** function multiplies a floating-point number x by a power of 2 given by exp and returns x × 2^{exp}. The exponent (exp) can be a negative or a positive value. A range error may occur 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 */
```

localtime — *Local Time*

Syntax

#include <time.h>

struct tm *localtime(time_t *timer);

Defined in

localtime.c in rts.src

Description

The **localtime** function converts a calendar time represented as a value of type time_t into a broken-down time in a structure. The function returns a pointer to the structure representing the converted time.

For more information about the functions and types that the time.h header declares, refer to subsection 5.2.10, page 5-10.

log — *Natural Logarithm*

Syntax

#include <math.h>

double log(double x);

Defined in

log.c in rts.src

Description

The **log** function returns the natural logarithm of a real number x. A domain error occurs if x is negative; a range error occurs if x is 0.

Example

```
float x, y;

x = 2.718282;
y = log(x);          /* Return value = 1.0 */
```

log10 — *Common Logarithm*

Syntax

#include <math.h>

double log10(double x);

Defined in

log10.c in rts.src

Description

The **log10** function returns the base-10 logarithm (or common 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 */
```

ltoa — *Convert Long Integer to ASCII*

Syntax

#include <stdlib.h>

int ltoa(long n, char *buffer);

Defined in

ltoa.c in rts.src

Description

The **ltoa** function converts a long integer n to the equivalent ASCII string and writes it into buffer with a null terminator. If the input number n is negative, a leading minus sign is output. The ltoa function returns the number of characters placed in the buffer, not including the terminator.

Example

```
int i;

char s[10];
i = ltoa (-92993L, s); /* i = 6, s = "-92993"*/
```

malloc

Allocate Memory

Syntax

#include <stdlib.h>

void *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, defined in an uninitialized named section called .sysmem in memory.c. The linker sets the size of this section from the value specified by the –heap option. Default heap size is 1K words. For more information, refer to subsection 4.1.3, *Dynamic Memory Allocation*, on page 4-4.

Example

This example allocates free space for a structure.

```
struct xyz *p;
p = malloc (sizeof (struct xyz));
```

memchr

Find First Occurrence of Byte

Syntax

#include <string.h>

void *memchr(void *s, char 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 s 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. The memchr function is expanded inline when the –x option is used.

memcmp — *Memory Compare*

Syntax

#include <string.h>

int memcmp(void *s1, void *s2, size_t n);

Defined in

memcmp.c in rts.src

Description

The **memcmp** function compares the first n characters of the object that s2 points to with the object that s1 points to. The function returns one of the following values:

<	0	if *s1 is less than *s2.
	0	if *s1 is equal to *s2.
>	0	if *s1 is greater than *s2.

The memcmp function is similar to strncmp, except that the objects that memcmp compares can contain values of 0. The memcmp function is expanded inline when the –x option is used.

memcpy — *Memory Block Copy — Nonoverlapping*

Syntax

#include <string.h>

void *memcpy(void *s1, 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. The memcpy function is expanded inline when the –x option is used.

memmove — *Memory Block Copy — Overlapping*

Syntax

#include <string.h>

void *memmove(void *s1, 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

#include <string.h>

void *memset(void *s, int c, size_t n);

Defined in

memset.c in rts.src

Description

The **memset** function copies the value of c into the first n characters of the object that s points to. The function returns the value of s. The memset function is expanded inline when the –x option is used.

minit

Reset Dynamic Memory Pool

Syntax

#include <stdlib.h>

void 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.

Note: Accessing Objects After Calling the minit Function

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.**

The memory that minit uses is in a special memory pool or heap, defined in an uninitialized named section called .sysmem in memory.c. The linker sets the size of this section from the value specified by the –heap option. Default heap size is 1 K words. For more information, refer to subsection 4.1.3, Dynamic Memory Allocation, on page 4-4.

mktime — *Convert to Calendar Time*

Syntax

#include <time.h>

time_t ***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 a time value of type time_t. 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.

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 */

printf ("result is %s\n", wday[time_str.tm_wday]);
```

For more information about the functions and types that the time.h header declares, refer to subsection 5.2.10, on page 5-10.

modf — *Signed Integer and Fraction*

Syntax

#include <math.h>

double modf(double value, double *iptr);

Defined In

modf30.asm 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 part 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.          */
```

pow — *Raise to a Power*

Syntax

#include <math.h>

double pow(double x, double y);

Defined In

pow.c in rts.src

Description

The **pow** function returns x raised to the power y. A domain error occurs if x = 0 and y ≤ 0, or if x is negative and y is not an integer. A range error may occur if the result is too large to represent.

Example

```
double x, y, z;

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

qsort — *Array Sort*

Syntax

#include <stdlib.h>

void qsort(void *base, size_t n, size_t size, int (*compar) (void));

Defined in

qsort.c in rts.src

Description

The **qsort** function sorts an array of n 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(void *ptrl, 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.

The array sort will not work correctly if the CMP function fails to return the correct values for all three conditions.

Example

In the following example, a short list of integers is sorted with qsort.

```
#include  <stdlib.h>

int list[ ] = {3, 1, 4, 1, 5, 9, 2, 6};
int idiff (const void *, const void *);

main( )
{
   qsort (list, 8, 1, idiff);
   /* after sorting, list[ ]=={ 1, 1, 2, 3, 4, 5, 6, 9} */
}

int idiff (const void *il, const void *i2)
{
   return *(int *)il - *(int *)i2;
}
```

rand/srand

Random Integer

Syntax

#include <stdlib.h>

int rand(void);
void srand(unsigned int seed);

Defined In

rand.c in rts.src

Description

Two functions work together to provide pseudorandom sequence generation:

- The **rand** function returns pseudorandom integers in the range 0—RAND_MAX. For the TMS320 floating-point C compiler, the value of RAND_MAX is 2147483646 (2^{31} –2).
- The **srand** function sets the value of the random number generator 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

#include <stdlib.h>

void *realloc(void *ptr, size_t size);

Defined In

memory.c in rts.src

Description

The **realloc** function changes the size of the allocated memory pointed to by ptr, 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 ptr is 0, realloc behaves like malloc.
- If ptr points to unallocated space, the function takes no action and returns.
- If the space cannot be allocated, the original memory space is not changed, and realloc returns 0.
- If size = 0 and ptr is not null, realloc frees the space that ptr points to.

If, in order to allocate more space, the entire object must be moved, realloc returns a pointer to the new space. Any memory freed by this operation is deallocated. If an error occurs, the function returns a null pointer (0).

The memory that realloc uses is in a special memory pool or heap, defined in an uninitialized named section called .sysmem in memory.c. The linker sets the size of this section from the value specified by the –heap option. Default heap size is 1K words. For more information, refer to subsection 4.1.3, *Dynamic Memory Allocation*, on page 4-4.

setjmp/longjmp *Nonlocal Jumps*

Syntax

#include <setjmp.h>

int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int returnval);

Defined in

setjmp.asm in rts.src

Description

The setjmp.h header defines one type, one macro, and one function for bypassing the normal function call and return discipline:

- The **jmp_buf** type is an array type suitable for holding the information needed to restore a calling environment.
- The **setjmp** macro saves its calling environment in the jmp_buf argument for later use by the longjmp function.

  If the return is from a direct invocation, the setjmp macro returns the value zero. 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 returnval. The longjmp function will not cause setjmp to return a value of zero even if returnval is zero. If returnval is zero, 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);
   . . .
}
```

sin

Sine

Syntax

#include <math.h>

double sin(double x);

Defined in

sin.c in rts.src

Description

The **sin** function returns the sine of a floating-point number x. x is an angle 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);      /* sin returns -1.0 */
```

sinh

Hyperbolic Sine

Syntax

#include <math.h>

double 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 */
```

sqrt

Square Root

Syntax

#include <math.h>

double sqrt(double x);

Defined in

sqrt30.asm in rts.src

Description

The **sqrt** function returns the non-negative 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 */
```

strcat

Concatenate Strings

Syntax

#include <string.h>

char *strcat(char *s1, char *s2);

Defined in

strcat.c in rts.src

Description

The **strcat** function appends a copy of s2 (including the terminating null character) to the end of s1. The initial character of s2 overwrites the null character that originally terminated s1. The function returns the value of s1. The strcat function is expanded inline if the –x option is used.

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"      */
strcat (a,c);
/* a —> "The quick black fox jumps over the lazy dog.\0"*/
```

strchr

Find First Occurrence of a Character

Syntax

#include <string.h>

char *strchr(char *s, char c);

Defined in

strchr.c in rts.src

Description

The **strchr** function finds the first occurrence of c in s. If strchr finds the character, it returns a pointer to the character; otherwise, it returns a null pointer (0). The strchr function is expanded inline if the –x option is used.

Example

```
char *a = "When zz comes home, the search is on for z's.";
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

#include <string.h>

int strcmp(char *s1, char *s2);
int strcoll(char *s1, char *s2);

Defined in

strcmp.c in rts.src

Description

The **strcmp** and **strcoll** functions compare s2 with s1. The functions are equivalent; both functions are supported to provide compatibility with ANSI C. The strcmp function is expanded inline if the –x option is used.

The functions return one of the following values:

<	0	if *s1 is less than *s2.
	0	if *s1 is equal to *s2.
>	0	if *s1 is greater than *s2.

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

#include <string.h>

char *strcpy(char *s1, char *s2);

Defined In

strcpy.c in rts.src

Description

The **strcpy** function copies s2 (including the 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. The strcpy function is expanded inline if the –x option is used.

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

#include <string.h>

size_t strcspn(char *s1, char *s2);

Defined In

strcspn.c in rts.src

Description

The **strcspn** function returns the length of the initial segment of s1, *which is made up entirely of characters that are not in s2.* If the first character in s1 is in s2, 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

```
#include <string.h>
```

char *strerror(int errnum);

Defined in

strerror.c in rts.src

Description

The **strerror** function returns the string error. This function is supplied to provide ANSI compatibility.

strftime *Format Time*

Syntax

#include <time.h>

size_t *strftime(char *s, size_t maxsize, char *format,
    struct tm *timeptr);

Defined in

strftime.c in rts.src

Description

The **strftime** function formats a time (pointed to by timeptr) according to a format string and returns the formatted result in the string s. Up to maxsize characters can be written to s. The format parameter is a string of characters that tells the strftime function how to format the time. The following list shows the valid characters and describes what each character expands to.

Character	***is replaced by ...***
`%a`	the abbreviated weekday name (Mon, Tue, . . .)
`%A`	the full weekday name.
`%b`	the abbreviated month name (Jan, Feb, . . .)
`%B`	the locale's full month name
`%c`	the date and time representation
`%d`	the day of the month as a decimal number (0–31)
`%H`	the hour (24-hour clock) as a decimal number (00–23)
`%I`	the hour (12-hour clock) as a decimal number (01–12)
`%j`	the day of the year as a decimal number (001–366)
`%m`	the month as a decimal number (01–12)
`%M`	the minute as a decimal number (00–59)
`%p`	the locale's equivalent of either A.M. or P.M.
`%S`	the second as a decimal number (00–50)
`%U`	the week number of the year (Sunday is the first day of the week) as a decimal number (00–52)
`%x`	the date representation
`%X`	the time representation
`%y`	the year without century as a decimal number (00–99)
`%Y`	the year with century as a decimal number
`%Z`	the time zone name, or by no characters if no time zone exists

For more information about the functions and types that the time.h header declares, refer to subsection 5.2.10, page 5-10.

strlen — *Find String Length*

Syntax

#include <string.h>

size_t strlen(char *s);

Defined In

strlen.c in rts.src

Description

The **strlen** function returns the length of s. In C, a character string is terminated by the first byte with a value of 0 (a null character). The returned result does not include the terminating null character. The strlen function is expanded in-line if the –x option is used.

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

#include <string.h>

char *strncat(char *s1, char *s2, size_t n);

Defined In

strncat.c in rts.src

Description

The **strncat** function appends up to n characters of s2 (including the terminating null character) to the end of s1. The initial character of s2 overwrites the null character that originally terminated s1; strncat appends a null character to the result. The function returns the value of s1.

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"   */;

strncat (a,c,size);

/* a—> "I do not like them, Sam I am, I do not like\0"   */;
```

strncmp — *Compare Strings*

Syntax

#include <string.h>

int strncmp(char *s1, char *s2, size_t n);

Defined in

strncmp.c in rts.src

Description

The **strncmp** function compares up to n characters of s2 with s1. The function returns one of the following values:

<	0	if *s1 is less than *s2.
	0	if *s1 is equal to *s2.
>	0	if *s1 is greater than *s2.

Example

```
char *stra = "why ask why";
char *strb = "just do it";
char *strc = "why not?";
size_t size = 4;

if (strcmp(stra, strb, size) > 0)
   {
      /* statements here will get executed */
   }
if (strcmp(stra, strc, size) == 0)
   {
      /* statements here will get executed also */
   }
```

strncpy *String Copy*

Syntax

#include <string.h>

char *strncpy(char *s1, char *s2, size_t n);

Defined In

strncpy.c in rts.src

Description

The **strncpy** function copies up to n characters from s2 into s1. If s2 is n characters long or longer, the null character that terminates s2 is not copied. If you attempt to copy characters from overlapping strings, the function's behavior is undefined. If s2 is shorter than n characters, strncpy appends null characters to s1 so that s1 contains n characters. The function returns the value of s1.

Example

Note that strb contains a leading space to make it five characters long. Also note that the first five characters of strc are an "I", a space, the word "am", and another space, so that after the second execution of strncpy, stra begins with the phrase "I am" followed by two spaces. In the comments, the notation "\0" represents the null character.

```
char *stra = "she's the one mother warned you of";
char *strb = " he's";
char *strc = "I am the one father warned you of";
char *strd = "oops";
size_t length = 5;

strncpy (stra,strb,length);

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

strncpy (stra,strc,length);

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

strncpy (stra,strd,length);

/* stra—> "oops\0"                               */;
/* strb—> " he's\0"                              */;
/* strc—> "I am the one father warned you of\0"  */;
/* strd—> "oops\0"                               */;
```

strpbrk

Find Any Matching Character

Syntax

#include <string.h>

char *strpbrk(char *s1, char *s2);

Defined In

strpbrk.c in rts.src

Description

The **strpbrk** function locates the first occurrence in s1 of *any* character in s2. 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

#include <string.h>

char *strrchr(char *s, int c);

Defined In

strrchr.c in rts.src

Description

The **strrchr** function finds the last occurrence of c in s. If strrchr finds the character, it returns a pointer to the character; otherwise, it returns a null pointer (0). The strrchr function is expanded inline if the –x option is used.

Example

```
char *a = "When zz comes home, the search is on for z's";
char *b;
char the_z = 'z';
     b = strrchr(a,the_Z);
```

After this example, b points to the z in zs near the end of the string.

strspn

Find Number of Matching Characters

Syntax

#include <string.h>

size_t *strspn(int *s1, int *s2);

Defined in

strspn.c in rts.src

Description

The **strspn** function returns the length of the initial segment of s1, *which is entirely made up* of characters in s2. If the first character of s1 is not in s2, 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

#include <string.h>

char *strstr(char *s1, char *s2);

Defined in

strstr.c in rts.src

Description

The **strstr** function finds the first occurrence of s2 in s1 (excluding the terminating null character). If strstr finds the matching string, it returns a pointer to the located string; if it doesn't find the string, it returns a null pointer. If s2 points to a string with length 0, strstr returns s1.

Example

```
char *stra = "so what do you want for nothing?";
char *strb = "what";
char *ptr;

ptr = strstr(stra,strb);
```

The pointer ptr now points to the "w" in "what" in the first string.

strtod/strtol/strtoul

Convert String to Numeric Value

Syntax

#include <stdlib.h>

double strtod(char *nptr, char **endptr);

long int strtol(char *nptr, char **endptr, int base);

unsigned long int strtoul(char *nptr, char **endptr, int base);

Defined in

strtod.c in rts.src, strtol.c in rts.src and strtoul.c in rts.src

Description

Three functions convert ASCII strings to numeric values. For each function, argument nptr 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 on which base to interpret the string.

- The **strtod** function converts a string to a floating-point value. The string must have the following format:

  [space] [sign] digits [.digits] [e|E [sign] integer]

  The function returns the converted string; if the original string is empty or does not have the correct format, the function returns a 0. If the converted string would cause an overflow, the function returns $_{+}$HUGE_VAL; if the converted string would cause an underflow, the function returns 0. If the converted string causes an overflow or an underflow, errno is set to the value of ERANGE.

- The **strtol** function converts a string to a long integer. The string must have the following format:

  [space] [sign] digits [.digits] [e|E [sign] integer]

- The **strtoul** function converts a string to an unsigned long integer. The string must be specified in the following format:

  [space] [sign] digits [.digits] [e|E [sign] integer]

The *space* is indicated by one or more of the following characters: space bar, horizontal or vertical tab, carriage return, form feed, or newline. Following the space is an optional *sign*, and then *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

#include <string.h>

char *strtok(char *s1, char *s2);

Defined in

strtok.c in rts.src

Description

Successive calls to the **strtok** function break s1 into a series of tokens, each delimited by a character from s2. Each call returns a pointer to the next token. The first call to strtok uses the string s1. Successive calls use a null pointer as the first argument. The value of s2 can change at each invocation. It is important to note that s1 is altered by the strtok function.

Example

After the first invocation of strtok in the example below, the pointer stra points to the string “excuse” 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"     */
```

tan — *Tangent*

Syntax

#include <math.h>

double tan(double x);

Defined in

tan.c in rts.src

Description

The **tan** function returns the tangent of a floating-point number x. x is an angle expressed in radians. An argument with a large magnitude may 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

#include <math.h>

double tanh(double x);

Defined in

tanh.c in rts.src

Description

The **tanh** function returns the hyperbolic tangent of a floating-point number x.

Example

```
double x, y;

x = 0.0;
y = tanh(x);        /* return value = 0.0 */
```

time

Time

Syntax

#include <time.h>

time_t time(time_t *timer);

Defined in

time.c in rts.src

Description

The **time** function determines the current calendar time, represented as a value of type time_t. The value is the number of seconds since 12:00 A.M., Jan 1, 1900. If the calendar time is not available, the function returns –1. If timer is not a null pointer, the function also assigns the return value to the object that timer points to.

For more information about the functions and types that the time.h header declares, refer to subsection 5.2.10, page 5-10.

Note: Writing Your Own Time Function

The time function is target-system specific, so you must write your own time function.

toascii

Convert to ASCII

Syntax

#include <ctype.h>

int toascii(char 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, _toascii.

tolower/toupper

Convert Case

Syntax

#include <ctype.h>

int tolower(char c);
int toupper(char c);

Defined in

tolower.c in rts.src
toupper.c in rts.src

Description

Two functions convert the case of a single alphabetic character, c, to upper or lower case:

- The **tolower** function converts an uppercase argument to lowercase. If c is not an uppercase letter, tolower returns it unchanged.
- The **toupper** function converts a lowercase argument to uppercase. If c is not a lowercase letter, toupper returns it unchanged.

The functions have macro equivalents named _tolower and _toupper.

Example

```
tolower ('A')        /* returns 'a' */
tolower ('+')        /* returns '+' */
```

va_arg/va_end/va_start — *Variable-Argument Macros/Functions*

Syntax

#include <stdarg.h>

type	**va_arg(ap,** *type***);**
void	**va_end(ap);**
void	**va_start(ap,** *parmN***);**
va_list	***ap;**

Defined In

stdarg.h as macros

Description

Some functions can be called with a varying number of arguments that have varying types. Such functions, called *variable-argument functions*, can use the following macros to step through argument lists at runtime. The ap parameter points to an argument in the variable-argument list.

- ❑ The **va_start** macro initializes ap to point to the first argument in an argument list for the variable-argument function. The parmN parameter points to the rightmost parameter in the fixed, declared list.
- ❑ The **va_arg** macro returns the value of the next argument in a call to a variable-argument function. Each time you call va_arg, it modifies ap so that successive arguments for the variable-argument function can be returned by successive calls to va_arg (va_arg modifies ap to point to the next argument in the list). The *type* parameter is a type name; it is the type of the current argument in the list.
- ❑ The **va_end** macro resets the stack environment after va_start and va_arg are used.

Note that you must call va_start to initialize ap before calling va_arg or va_end.

Example

error:
Getting Started
Guide: 2.4

```
int printf (char *fmt, ...)


{
   va_list ap;
   va_start(ap, fmt);
       .
       .
       .
/* Get next arg, an integer      */
   i = va_arg(ap, int);
/* Get next arg, a string        */
   s = va_arg(ap, char *);
/* Get next arg, a long          */
   l = va_arg(ap, long);
       .
       .
       .
   va_end(ap)       /* Reset              */
}
```

Chapter 6

Library-Build Utility

When using the TMS320 floating-point C compiler, you can compile your code under a number of different configurations and options, which are not necessarily compatible with one another. Since it would be cumbersome to include all possible combinations in individual runtime-support libraries, this package includes the source file, rts.src, that contains all runtime-support functions, and all floating-point support functions. By using the mk30 utility described in this chapter you can custom build your own runtime-support libraries for the options you select.

These are the topics covered in this chapter:

Topic **Page**

6.1 Invoking the Library-build utility

The general syntax for invoking the library utility is:

```
mk30 [options] src_arch1 [–lobj.lib1] [src_arch2 [–lobj.lib2]]
```

mk30	is the command that invokes the utility.
options	can appear anywhere on the command line or in a command file. (Options are discussed in Section 6.2 and below.)
src_arch	is the name of a source archive file. For each source archive named, mk30 will build an object library according to the runtime model specified by the command line options.
–l*obj.lib*	is the optional object library name. If you do not specify a name for the library, mk30 uses the name of the source archive and appends a *.lib* suffix. For each source archive file specified, a corresponding object library file is created. An object library cannot be built from multiple source archive files.

The mk30 utility runs the shell program cl30 on each source file in the archive to either compile or assemble it. It then collects all the object files into the output library. All the tools must be in your PATH. *The utility ignores and disables the environment variables* TMP, C_OPTION, *and* C_DIR.

Library Utility Specific Options

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

– –c	extracts C source files contained in the source archive from the library and leaves them in the current directory after the utility has completed execution.
– –h	instructs mk30 to use header files contained in the source archive and leave them in the current directory after the utility has completed execution. You will probably want to use this option to install the runtime-support header files from the rts.src archive that is shipped with the tools.
– –k	instructs the mk30 utility to over-write 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.

––u instructs mk30 not to use the header files contained in the source archive when building the object library. If the desired headers are already in the current directory, there is no reason to reinstall them. This option also gives you some flexibility in modifying runtime-support functions to suit your application.

––v prints progress information to the screen during execution of the utility. Normally, the utility operates silently (no screen messages).

Example: The following command builds the standard runtime support library as an object library named rts40r.lib. The library is compiled for the TMS320C4x (–v40), optimized with inline function expansion (–x and –o), and uses the register-argument runtime conventions. The example assumes that the runtime support headers already exist in the current directory (––u).

```
mk30 ––u –v40 –o –x –mr rts.src –l rts40r.lib
```

6.2 Options Summary

Options for the mk30 utility correspond directly to the options that the compiler uses. These options are described in detail in subsection 2.1.3 on page 2-5.

General Options	Effect
–g	symbolic debugging
–vxx	target processor TMS320Cxx
Parser Options	**Effect**
–pk	K&R compatible; compatible with previous C standards
–pw	suppress warning messages
–p?	enable trigraph expansion
Optimizer Options	**Effect**
–o0	level 0 register optimization
–o1	level 1 +local optimization
–o2 (or –o)	level 2 +global optimization
Inlining Options	**Effect**
–x0	disables inlining
–x1	default inlining level
–x2 (or –x)	defines _INLINE + invokes optimizer at level 2
Runtime Model Options	**Effect**
–ma	assumes aliased variables
–mb	big memory model
–mc	fast float to int conversion
–mf	far pointers
–mi	avoid RPTS loops
–mm	use MPYI for multiply
–mn	enables optimization disabled by –g
–mr	register argument conventions
–mx	avoid TMX silicon bugs
Type Checking Options	**Effect**
–tf	relax prototype checking
–tp	relax pointer combination checking
Assembler Options	**Effect**
–as	keep labels as symbols

Appendix A

Description of Compiler Optimizations

The TMS320 floating-point C compiler uses a variety of optimization techniques to improve the execution speed of your C programs and to reduce their size. Optimization occurs at various levels throughout the compiler. Most of the optimizations described here are performed by the separate optimizer pass that you enable and control with the –o compiler options. (For details about the –o options, refer to Section 2.4 on page 2-32.) However, the code generator performs some optimizations, particularly the target-specific optimizations, that you cannot selectively enable or disable.

This appendix describes two categories of optimizations: general and target-specific. General optimizations improve any C code, and target-specific optimizations are designed especially for the TMS320C3x/C4x architecture. Both kinds of optimizations are performed throughout the compiler. Some of the examples in this section were compiled with the –v40 option, and some with –v30. The code produced will be slightly different different depending on the version. For example, Example A–1 shows code produced with the –v40 option, so one of its local variables is assigned to register R8 (a register the 'C3x doesn't have). Unless otherwise noted, the differences between –v30 and –v40 code are minimal.

These are the optimizations covered in this appendix:

Target-Specific Optimizations

- Register variables
- Register tracking/targeting
- Cost-based register allocation
- Autoincrement addressing
- Repeat blocks
- Parallel instructions
- Conditional instructions
- Delayed branches
- TMS320C4x-specific features

General Optimizations

- Algebraic reordering, symbolic simplification, constant folding
- Alias disambiguation
- Copy propagation
- Common subexpression elimination
- Redundant assignment elimination
- Branch optimizations, control-flow simplification, delayed branches
- Loop induction variable optimizations, strength reduction
- Loop unrolling
- Loop rotation
- Loop invariant code motion
- Inline function expansion

Effective register use

The TMS320 floating-point DSPs have a large and versatile set of registers. The compiler is designed to make effective use of the registers, for both general computations and in cases where specific registers can be used in specialized ways.

❑ **Register variables**

The compiler helps maximize the use of registers for storing local variables, parameters, and temporary values. Variables stored in registers can be accessed more efficiently than variables in memory. Register variables are particularly effective for pointers. See Example A–1 and Example A–2.

❑ **Register tracking/targeting**

The compiler tracks the contents of registers so that it avoids reloading values if they are used again soon. Variables, constants, and structure references such as (a.b) are tracked through both straight-line code and forward branches. The compiler also uses register targeting to compute expressions directly into specific registers when required, as in the case of assigning to register variables or returning values from functions. See Example A–1.

Example A–1. Register Variables and Register Tracking/Targeting

```
    int gvar;
    reg(int i, int j)
    {
       gvar = call () & i;
       j    = gvar + i;


       return j;
}
    }

**********************************************************************
*   TMS320C30 C COMPILER                           Version X.XX      *
**********************************************************************
;      ac30 -mr a1.c a1.if
;      opt30 -r -02 a1.if a1.opt
;      cg30 -p a1.opt a1.asm a1.tmp
       .version          30
FP     .set              AR3
       .globl  _gvar
       .globl  _reg
**********************************************************************
*   FUNCTION DEF : _reg                                              *
**********************************************************************
_reg:
       PUSH  R4
*
* R4   assigned to parameter i
*
       LDI   AR2,R4
       CALL  _call
       AND   R4,R0,R2
       STI   R2,@_gvar
       ADDI  R4,R2,R0
EPIO_1:
       POP   R4
       RETS
       .globl _gvar
       .bss   _gvar,1
**********************************************************************
*   UNDEFINED REFERENCES                                             *
**********************************************************************
       .globl _call
       .end
```

- **Cost-based register allocation**

  The optimizer, when enabled, allocates registers to user variables and compiler temporary values according to their type, use, and frequency. Variables used within loops are weighted to have priority over others, and those variables whose uses don't overlap may be allocated to the same register. Variables with specific requirements are allocated into registers that can accommodate them. For example, the compiler tries to allocate loop counters into RC. The allocation algorithm also minimizes pipeline conflicts that can result from performing calculations using address generation registers in the vicinity of indirection.

- **Passing arguments in registers**

  The compiler supports a new, optional calling sequence that passes arguments to registers rather than pushing them onto the stack. This can result in significant improvement in execution performance, especially if calls are important in the application. See Example A–5.

Autoincrement addressing

For pointer expressions of the form `*p++`, `*p––`, `*++p`, or `*––p`, the compiler uses efficient TMS320C3x/C4x 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 A–2.

Repeat blocks

The TMS320C3x/C4x supports zero-overhead loops with the RPTS (repeat single) and RPTB (repeat block) instructions. With the optimizer, the compiler can detect loops controlled by counters and generate code for them using the efficient repeat forms: RPTS for single-instruction loops, or RPTB for larger loops. For both forms, the iteration count can be either a constant or an expression. See Example A–2 and Example A–8.

Parallel instructions

Several TMS320C3x/C4x instructions such as load/load, store/operate, and multiply/add can be paired with each other and executed in parallel. When adjacent instructions match the addressing requirements, the compiler combines them in parallel. Although the code generator performs this optimization, the optimizer greatly increases effectiveness because operands are more likely to be in registers. See Example A–2 and Example A–8.

Example A–2. Repeat Blocks, Autoincrement Addressing, Parallel Instructions, Strength Reduction, Induction Variable Elimination, Register Variables, and Loop Test Replacement

```
float a[10], b[10];
float dot_product ()
{
   int i;
   float sum;
   for (i=0; i<10; i++)
      sum += a[i] * b[i];
   return sum
}
**********************************************************************
*  TMS320C30 C COMPILER                               Version X.XX      *
**********************************************************************
;      ac30 -mr a2.c a2.if
;      opt30 -r -02 a2.if a2.opt
;      cg30 -p a2.opt a2.asm a2.tmp
       .version         30
FP     .set             AR3
       .globl  _a
       .globl _b
       .globl  _dot_product
**********************************************************************
*  FUNCTION DEF : _dot_product                                          *
**********************************************************************
_dot_product:
       PUSH  AR4
*
* R2   assigned to variable sum
*
       LDI    @CONST+0,AR4
       LDI    @CONST+1,AR2
       MPYF   *AR2++,*AR4++,R0
       RPTS   8
       ADDF   R0,R2
    || MPYF   *AR2++,*AR4++,R0
       ADDF   R0,R2
       LDF    R2,R0
EPI0_1:
       POP    AR4
       RETS
       .globl _a
       .bss   _a,10
       .globl _b
       .bss   _b,10
**********************************************************************
*  DEFINE CONSTANTS                                                     *
**********************************************************************
       .bss   CONST,2
       .sect  ".cinit"
       .word  2,CONST
    .  .word  _a                                        ;0
       .word  _b                                        ;1
       .end
```

Conditional Instructions

The load instructions in the TMS320C3x/C4x can be executed conditionally. For simple assignments such as

`a = condition ? exprl : expr2 or if (condition) a = b;`

the compiler can use conditional loads to avoid costly branches.

Delayed branches

The TMS320C3x/C4x provides delayed branch instructions that can be inserted three instructions early in an instruction stream, avoiding costly pipeline flushes associated with normal branches. The compiler uses unconditional delayed branches wherever possible, and conditional delayed branches for counting loops. See Example A–3.

Example A–3. TMS320C3x/C4x Delayed Branch Instructions

```
     wait(volatile int *p)
     {
           for(;;)

                if (*p & 0x08) *p |= 0xF0
     }

**********************************************************************
*    TMS320C30 C COMPILER                               Version X.XX *
**********************************************************************
;          ac30 -mr a3.c a3.if
;          opt30 -r -02 a3.if a3.opt
;          cg30 -p a3.opt a3.asm a3.tmp
           .version        30
FP         .set            AR3
           .globl _wait
**********************************************************************
*    FUNCTION DEF : _wait                                            *
**********************************************************************
_wait
*
*    AR2   assigned to parameter p
*
L2
           LDI  8,R0
           TSTB R0.*AR2
           BZ   L2
           LDI  240,R1
           OR   R1,*AR2,R2
           STI  R2.*AR2
***        B    L2       ;BRANCH OCCURS
           .end
```

TMS320C4x-specific features

The TMS320C4x has several instructions and addressing modes that are particularly useful in compiled code. The instructions include store integer immediate (STIK), load address register (LDA), load data page pointer (LDPK), and a 32×32 integer multiply (MPYI). The new addressing modes allow constants and indirect+offset operands to be used in 3-operand instructions. In addition, separate AR interlocks eliminate the pipeline delays associated with independent use of the AR registers. Example A–4 illustrates some of the new 'C4x features.

Example A–4. TMS320C4x-Specific Features

```
typedef struct S {int i, j, k; } s;
sval (s *p)
{
     p–>j = 5;
     p–>k *= 10;
     return p–>i;
}
************************************************************************
* TMS320C30 C COMPILER                                 Version X.XX    *
************************************************************************
;          ac30 –v40 a4.c a4.if
;          opt30 –v40 –02 a4.if a4.opt
;          cg30 –v40 a4.opt a4.asm a4.tmp
           .version             40
FP         .set                 AR3
           .globl  _sval
************************************************************************
* FUNCTION DEF : _sval                                                 *
************************************************************************
_sval
           PUSH      FP
           LDI       SP, FP
*
* AR2      assigned to parameter p
*
           LDA       *–FP,(2),AR2
           STIK      5,*+AR2(1)
           MPYI      10,*+AR2(2),R0
           STI       R0,*+AR2(2)
           LDI       *AR2,R0
EPIO_1:
           LDI       *–FP(1),R1
           BD        R1
           LDI       *FP,FP
           NOP
           SUB1      2,SP
           B         R1         ;BRANCH OCCURS
           .end
```

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) requires 5 instructions and 2 registers to evaluate; it can be optimized to ((a + b) – c) – d, which takes only 4 instructions and 1 register. Operations between constants are folded into single constants. For example, a = (b + 4) – (c + 1) becomes a = b – c + 3. See Example A–5.

Alias disambiguation

Programs written in the C language generally use many pointer variables. Frequently, compilers are unable to determine whether or not two or more l (lowercase L) values (symbols, pointer references, or structure references) refer to the same memory location. This *aliasing* of memory locations often prevents the compiler from retaining values in registers, because it cannot be sure that the register and memory continue to hold the same values over time. Alias disambiguation is a technique that determines when two pointer expressions cannot point to the same location, allowing the compiler to optimize such expressions.

Data-flow optimizations

Collectively, the following three data-flow optimizations replace expressions with less costly ones, detect and remove unnecessary assignments, and avoid operations that produce values already computed. The optimizer performs these data flow optimizations both locally (within basic blocks) and globally (across entire functions). See Example A–5 and Example A–6.

- **Copy propagation**

  Following an assignment to a variable, the compiler replaces references to the variable with its value. The value could be another variable, a constant, or a common subexpression. This may result in increased opportunities for constant folding, common subexpression elimination, or even total elimination of the variable. See Example A–5 and Example A–6.

- **Common subexpression elimination**

  When the same value is produced by two or more expressions, the compiler computes the value once, saves it, and reuses it. See Example A–5.

- **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 A–5.

Example A–5. Data-Flow Optimizations

```
    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);
    }
************************************************************************
* TMS320C30 C COMPILER                                  Version X.XX   *
************************************************************************
;       ac30 -mr a5.c a5.if
;       opt30 -r -02 a5.if a5.opt
;       cg30 -p a5.opt a5.asm a5.tmp
    .   version                                         30
FP      .set                                            AR3
        .globl  _simp
************************************************************************
* FUNCTION DEF : _simp                                                 *
************************************************************************
_simp
*
* AR2  assigned to parameter j
*
        LDI     2,R0
        LSH     R0,AR2,R1
        ADDI    R1,AR2,R2
        LDI     3,R1
        LSH     R1,AR2,R3
        LSH     R2,AR2,RC
        ADDI    R3,RC
        LDI     3,AR2
        CALL    _call
EPIO_1:
        RETS
************************************************************************
* UNDEFINED REFERENCES                                                 *
************************************************************************
        .globl _call
        .end
```

Branch optimizations / 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 can be determined at compile time (through copy propagation or other data flow analysis), a conditional branch can be deleted. 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. See Example A–6.

Example A–6. Copy Propagation and Control-Flow Simplification

```
     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;
         }
     }

**************************************************************************
* TMS320C40 C COMPILER                                      Version X.XX*
**************************************************************************
;      ac30 –v40 a6.c a6.if
;      opt30 –v40 –02 a6.if a6.opt
;      cg30 –v40 a6.opt a6.asm a6.tmp
       .version      40
FP     .set    AR3
    .globl   _fsm
**** *********************************************************************
* FUNCTION DEF : _fsm                                                    *
**************************************************************************
_fsm
*
* AR2  assigned to variable input
*
       LDI  *AR2++,R0
       BNZ L4
L2:
       LDI *AR2++,R0
       BZ  L4
       LDI AR2++,R0
       BZ  L2
L4:
       LDI *AR2++,R0
       BZ  L4
EPIO_1:
       RETS
       .end
```

Loop induction variable optimizations / strength reduction

Loop 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 costly 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. Loops controlled by incrementing a counter are written as TMS320C3x/C4x repeat blocks or by using efficient decrement-and-branch instructions. Induction variable analysis and strength reduction together often remove all references to your loop control variable, allowing it to be eliminated entirely. See Example A–2 and Example A–8.

Loop Unrolling

When the compiler can determine that a short loop is executed a low, constant number of times, it replicates the body of the loop rather than generating the loop. This avoids any branches or use of the repeat registers. ("Low" and "short" are subjective judgments made by the compiler.) See Example A–7.

Example A–7. Loop Unrolling

```
      add3(int a[3])
      {
          int i, sum = 0;
          for (i=0; i<3; i++) sum += a[i]
      return sum;
      }
***********************************************************************
*     TMS320C30 C COMPILER                              Version   X.XX*
***********************************************************************
;         ac30 a7.c a7.if
;         opt30 -02 a7.if a7.opt
;         cg30 a7.opt a7.asm a7.tmp
          .version          30
FP        .set              AR3
          .globl   _add3
***********************************************************************
*    FUNCTION DEF : _add3                                            *
***********************************************************************
_add3:
          PUSH    FP
          LDI     SP, FP
*
* R2  assigned to variable sum
*
          LDI     *-FP(2),AR2
          LDI     0,R2
          ADDI    *AR2++,R2
          ADDI    *AR2++,R2
          ADDI    *AR2,R2
          LDI     R2,R0
EPIO_1:
          LDI     *-FP(1),R1
          BD      R1
          LDI     *FP,FP
          NOP
          SUBI    2,SP
***       B       R1        ;BRANCH OCCURS
          .end
```

Loop rotation

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

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 A–8.

Inline function expansion

The special keyword inline directs the compiler to replace calls to a function with inline code, saving the overhead associated with a function call as well as providing increased opportunities to apply other optimizations. See Example A–8 and Example A–9. The optimizer inlines small functions not declared as inline when invoked with the –o3 option. The size of code growth allowed may be changed using the –oi *size* option.

Example A–8. Inline Function Expansion, part one

```
inline blkcpy (char *to, char *from, int n)
{
   if (n > 0 )
      do *to++ = *from++; while (——n != 0);
}
struct s {int a , b, c[10]; };
initstr (struct s *ps, char t[12])
{
   blkcpy((char *)ps, t, 12);
}
```

Example A–9. Inline Function Expansion, part two

```
***********************************************************************
*  TMS320C30 C COMPILER                                 Version X.XX     *
***********************************************************************
;      ac30 -mr a8.c a8.if
;      opt30 -r -02 a8.if a8.opt
;      cg30 -p a8.opt a8.asm a8.tmp
       .version         30
FP     .set             AR3
       .globl  _blkcpy
***********************************************************************
*  FUNCTION DEF : _blkcpy                                              *
***********************************************************************
_blkcpy:
       PUSH   AR4
*
* R3   assigned to parameter n
* AR2  assigned to parameter to
* AR4  assigned to parameter from
*
       LDI    R2,AR4
       CMPI   0,R3
       BLE    EPI0_1
       LDI    R3,RC
       SUBI   1,RC
       RPTB   L5
       LDI    *AR4++,R0
L5:    STI    R0,*AR2++
EPI0_1:
       POP    AR4
       RETS
       .globl _initstr
***********************************************************************
*  FUNCTION DEF : _initstr                                             *
***********************************************************************
_initstr
       PUSH   AR4
*
* AR2  assigned to parameter ps
* AR2  assigned to variable to
* AR4  assigned to parameter to
* AR4  assigned to variable from
*
       LDI    R2,AR4
* >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>ENTERING blkcpy()
       LDI    *AR4++,R0
       RPTS   10
       STI    R0,*AR2++
    || LDI    *AR4++,R0
       STI    R0,*AR2++
EPI0_2:
       POP    AR4
       RETS
       .end
```

Appendix B

Glossary

A

aliasing: a method of accessing a single data object in more than one way, as when a pointer points to a named object. The optimizer has logic to detect aliasing, but aliasing can cause problems for the optimizer.

allocation: A process in which the linker calculates the final memory addresses of output sections.

archive library: A collection of individual files that have been grouped into a single file.

archiver: A software program that allows you to collect several individual files into a single file called an archive library. The archiver also allows you to delete, extract, or replace members of the archive library, as well as to add new members.

assembler: A software program that creates a machine-language program from a source file that contains assembly language instructions, directives, and macro directives. The assembler substitutes absolute operation codes for symbolic operation codes, and absolute or relocatable addresses for symbolic addresses.

assignment statement: A statement that assigns a value to a variable.

autoinitialization: The process of initializing global C variables (contained in the .cinit section) before beginning program execution.

B

binding: Associating or linking together two complementary software objects.

block: A set of declarations and statements that are grouped together with braces.

.bss: One of the default COFF sections. You can use the .bss directive to reserve a specified amount of space in the memory map that can later be used for storing data. The .bss section is uninitialized.

C

C compiler: A program that translates C source statements into TMS320 floating-point assembly language source statements.

COFF (common object file format): A binary object file format that promotes modular programming by supporting the concept of *sections.*

comment: A source statement (or portion of a source statement) that is used to document or improve readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.

constant: A numeric value that can be used as an operand.

cross-reference lister: A debugging tool that accepts linked object files as input and produces cross-reference listings as output.

cross-reference listing: An output file created by the assembler that lists the symbols that were defined, what line they were defined on, which lines referenced them, and their final values.

D

.data: One of the default COFF sections. The .data section is an initialized section that contains initialized data. You can use the .data directive to assemble code into the .data section.

directive: Special-purpose commands that control the actions and functions of a software tool (as opposed to assembly language instructions, which control the actions of a device).

E

environment variables: System symbols that you define and assign to a string. They are usually included in various batch files; for example, in AUTOEXEC.BAT.

entry point: The starting execution point in target memory.

executable module: An object file that has been linked and can be executed in a TMS320 system.

expression: A constant, a symbol, or a series of constants and symbols separated by arithmetic operators.

F

field: For the TMS320, a software-configurable data type whose length can be programmed to be any value in the range of 1–16 bits.

G

global: A kind of symbol that is either 1) defined in the current module and accessed in another, or 2) accessed in the current module but defined in another.

H

high-level language debugging: The ability of a compiler to retain symbolic and high-level language information (such as type and function definitions) so that a debugging tool can use this information.

hole: An area between the input sections that compose an output section that contains no actual code or data.

I

initialized section: A COFF section that contains executable code or initialized data. An initialized section can be built up with the .data, .text, or .sect directive.

input section: A section from an object file that will be linked into an executable module.

K

K & R: *The C Programming Language* (second edition), by Brian Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey, 1988. This book describes ANSI C and is used as a reference in this book. Paragraphs within the book are referred to with this symbol: §.

L

label: A symbol that begins in column 1 of a source statement and corresponds to the address of that statement.

linker: A software tool that combines object files to form an object module that can be allocated into TMS320 system memory and executed by the TMS320.

listing file: An output file, created by the assembler, that lists source statements, their line numbers, and their effects on the SPC.

loader: A device that loads an executable module into TMS320 system memory.

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 source statements that are substituted for the macro call and are subsequently assembled.

map file: An output file, created by the linker, that shows the memory configuration, section composition, and section allocation, as well as symbols and the addresses at which they were defined.

member: The elements or variables of a structure, union, archive, or enumeration.

memory map: A map of TMS320 target system memory space, which is partitioned off into functional blocks.

O

object file: A file that has been assembled or linked and contains machine-language object code.

object library: An archive library made up of individual object files.

operand: The arguments, or parameters, of an assembly language instruction, assembler directive, or macro directive.

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

protected mode: Thirty-two bit extended DOS mode. These programs require an extended memory manager and will run only on larger processors ('386 or better). They can utilize all the available RAM on the computer.

R

RAM model: An autoinitialization model used by the linker when linking C code. The linker uses this model when you invoke the linker with the –cr option. The RAM model allows variables to be initialized at load time instead of runtime.

real mode: Sixteen-bit native MS-DOS mode. This mode limits the available memory to 640K. Calls to DOS may involve switching from protected to real mode.

ROM model: An autoinitialization model used by the linker when linking C code. The linker uses this model when you invoke the linker with the –c option. In the ROM model, the linker loads the .cinit section of data tables into memory, and variables are initialized at runtime.

S

section: A relocatable block of code or data that will ultimately occupy contiguous space in the TMS320 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.

static: 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 re-entered.

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.

symbolic debugging: The ability of a software tool to retain symbolic information so that it can be used by a debugging tool such as a simulator or an emulator.

symbol table: A portion of a COFF object file that contains information about the symbols that are defined and used by the file.

U

uninitialized section: A COFF section that reserves space in the TMS320 memory map but that has no actual contents. These sections are built up with the .bss and .usect directives.

union: A variable that may hold (at different times) objects of different types and sizes.

unsigned: A kind of value that is treated as a positive number, regardless of its actual sign.

V

virtual memory : The ability for a program to use more memory than a computer actually has available as RAM. This is accomplished by using a swap file on disk to augment RAM. When RAM is not sufficient, part of the program is swapped out to a disk file until it is needed again. The combination of the swap file and available RAM is the virtual memory. The TMS320 floating-point tools use a memory extender to provide virtual memory management. This memory extender is not provided as an executable but is embedded in several of the object programs.

W

word: A 32-bit addressable location in target memory.

Index

NOTES

NOTES

NOTES

TI Worldwide Sales and Representative Offices

AUSTRALIA / NEW ZEALAND: Texas Instruments Australia Ltd.: Sydney [61] 2-910-3100, Fax 2-805-1186; Melbourne 3-696-1211, Fax 3-696-4446.
BELGIUM: Texas Instruments Belgium S.A./N.V.: Brussels [32] (02) 242 75 80, Fax (02) 726 72 76.
BRAZIL: Texas Instrumentos Electronicos do Brasil Ltda.: Sao Paulo [55] 11-535-5133.
CANADA: Texas Instruments Canada Ltd.: Montreal (514) 335-8392; Ottawa (613) 726-3201; Toronto (416) 884-9181.
DENMARK: Texas Instruments A/S: Ballerup [45] (44) 68 74 00.
FINLAND: Texas Instruments/OY: Espoo [358] (0) 43 54 20 33, Fax (0) 46 73 23.
FRANCE: Texas Instruments France: Velizy-Villacoublay Cedex [33] (1) 30 70 10 01, Fax (1) 30 70 10 54.
GERMANY: Texas Instruments Deutschland GmbH.: Freising [49] (08161) 80-0, Fax (08161) 80 45 16; Hannover (0511) 90 49 60, Fax (0511) 64 90 331; Ostfildern (0711) 34 03 0, Fax (0711) 34 032 57.
HONG KONG: Texas Instruments Hong Kong Ltd.: Kowloon [852] 956-7288, Fax 956-2200.
HUNGARY: Texas Instruments Representation: Budapest [36] (1) 269 8310, Fax (1) 267 1357.
INDIA: Texas Instruments India Private Ltd.: Bangalore [91] 80 226-9007.
IRELAND: Texas Instruments Ireland Ltd.: Dublin [353] (01) 475 52 33, Fax (01) 478 14 63.
ITALY: Texas Instruments Italia S.p.A.: Agrate Brianza [39] (039) 68 42.1, Fax (039) 68 42.912; Rome (06) 657 26 51.
JAPAN: Texas Instruments Japan Ltd.: Tokyo [81] 03-769-8700, Fax 03-3457-6777; Osaka 06-204-1881, Fax 06-204-1895; Nagoya 052-583-8691, Fax 052-583-8696; Ishikawa 0762-23-5471, Fax 0762-23-1583; Nagano 0263-33-1060, Fax 0263-35-1025; Kanagawa 045-338-1220, Fax 045-338-1255; Kyoto 075-341-7713, Fax 075-341-7724; Saitama 0485-22-2440, Fax 0425-23-5787; Oita 0977-73-1557, Fax 0977-73-1583.
KOREA: Texas Instruments Korea Ltd.: Seoul [82] 2-551-2800, Fax 2-551-2828.
MALAYSIA: Texas Instruments Malaysia: Kuala Lumpur [60] 3-230-6001, Fax 3-230-6605.
MEXICO: Texas Instruments de Mexico S.A. de C.V.: Colina del Valle [52] 5-639-9740.
NORWAY: Texas Instruments Norge A/S: Oslo [47] (02) 264 75 70.
PEOPLE'S REPUBLIC OF CHINA: Texas Instruments China Inc.: Beijing [86] 1-500-2255, Ext. 3750, Fax 1-500-2705.
PHILIPPINES: Texas Instruments Asia Ltd.: Metro Manila [63] 2-817-6031, Fax 2-817-6096.
PORTUGAL: Texas Instruments Equipamento Electronico (Portugal) LDA.: Maia [351] (2) 948 10 03, Fax (2) 948 19 29.
SINGAPORE / INDONESIA / THAILAND: Texas Instruments Singapore (PTE) Ltd.: Singapore [65] 390-7100, Fax 390-7062.
SPAIN: Texas Instruments España S.A.: Madrid [34] (1) 372 80 51, Fax (1) 372 82 66; Barcelona (3) 31 791 80.
SWEDEN: Texas Instruments International Trade Corporation (Sverigefilialen): Kista [46] (08) 752 58 00, Fax (08) 751 97 15.
SWITZERLAND: Texas Instruments Switzerland AG: Dietikon [41] 886-2-3771450.
TAIWAN: Texas Instruments Taiwan Limited: Taipei [886] (2) 378-6800, Fax 2-377-2718.
UNITED KINGDOM: Texas Instruments Ltd.: Bedford [44] (0234) 270 111, Fax (0234) 223 459.

UNITED STATES: Texas Instruments Incorporated: **ALABAMA:**
Huntsville (205) 430-0114; **ARIZONA:** Phoenix (602) 244-7800;
CALIFORNIA: Irvine (714) 660-1200; San Diego (619) 278-9600;
(408) 894-9000; Woodland Hills (818) 704-8100; **COLORADO:** A
(303) 368-8000; **CONNECTICUT:** Wallingford (203) 265-3807; FL
Orlando (407) 260-2116; Fort Lauderdale (305) 425-7820; Tamp
(813) 882-0017; **GEORGIA:** Atlanta (404) 662-7967; **ILLINOIS:** Ar
Heights (708) 640-2925; **INDIANA:** Indianapolis (317) 573-6400;
Kansas City (913) 451-4511; **MARYLAND:** Columbia (410) 312-7
MASSACHUSETTS: Boston (617) 895-9100; **MICHIGAN:** Detroit
(313) 553-1500; **MINNESOTA:** Minneapolis (612) 828-9300; NEW
Edison (908) 906-0033; **NEW MEXICO:** Albuquerque (505) 345-2
NEW YORK: Poughkeepsie (914) 897-2900; Long Island (516) 4
Rochester (716) 385-6770; **NORTH CAROLINA:** Charlotte (704)
Raleigh (919) 876-2725; **OHIO:** Cleveland (216) 765-7258; Dayto
(513) 427-6200; **OREGON:** Portland (503) 643-6758; PENNSYLV
Philadelphia (215) 825-9500; **PUERTO RICO:** Hato Rey (809) 75
TEXAS: Austin (512) 250-6769; Dallas (214) 917-1264; Houston
(713) 778-6592; **WISCONSIN:** Milwaukee (414) 798-1001.

North American Authorized Distributors

COMMERCIAL
Almac / Arrow
Anthem Electronics
Arrow / Schweber
Future Electronics (Canada)
Hamilton Hallmark
Marshall Industries
Wyle

MILITARY
Alliance Electronics
Future Electronics (
Hamilton Hallmark
Zeus, An Arrow Co

CATALOG
Allied Electronics
Arrow Advantage
Newark Electronics

OBSOLETE PRODUCTS
Rochester Electronics 508/462-9332

For Distributors outside North America, contact your local Sales Offic

TEXAS INSTRUMENTS