TMS320C54x Optimizing C Compiler User's Guide

Literature Number: SPRU103B
Manufacturing Part Number: 2617678-9761 revision D
March 1997

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

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

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

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

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Preface

Read This First

About This Manual

The *TMS320C54x Optimizing C Compiler User's Guide* explains how to use these compiler tools:

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

The TMS320C54x C compiler accepts American National Standards Institute (ANSI) standard C source code and produces assembly language source code for the TMS320C54x devices. This user's guide discusses the characteristics of the TMS320C54x 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 using this book, read the *TMS320C54x 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 TMS320C54x devices. This book is divided into three distinct parts:

- **Introductory information**, consisting of Chapter 1, provides an overview of the TMS320C54x development tools.
- **Compiler description**, consisting of Chapters 2, 3, 4, 5, 6, and 7, 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 TMS320C54x 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**, consisting of Appendices A, B, and C, provides supplementary information on TMS320C54x-specific optimizations and individual tools as well as a glossary.

Notational Conventions

This document uses the following conventions:

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

  Here is a sample of C code:

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

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** and parameters are in *italic typeface*. Portions of a syntax that are in **bold** must be entered as shown; portions of a syntax that are in *italics* describe the type of information that must be entered. Here is an example of command line syntax:

  lnk500 *filenames*

  lnk500 is a command. The command invokes the linker and has one parameter, indicated by *filenames*. When you invoke the linker, you supply the names of the files that the linker uses as input.

- 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 is an example of a command that has optional parameters:

  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.

- 'C54x is used throughout this manual to collectively refer to the TMS320C541, TMS320C542, TMS320C543, TMS320C544, TMS320C545, TMS320C546, TMS320C547, TMS320C548, TMS320C549, and TMS320C545LP devices.

- K&R is used throughout this manual to refer to the ANSI standard for C as defined by ANSI and described in Kernighan and Ritchie's *The C Programming Language* (second edition).

- The 0x notation designates a hexadecimal number.

Related Documentation From Texas Instruments

The following books describe the TMS320C54x and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C54x DSP Reference Set is composed of four volumes that can be ordered as a set with literature number SPRU210. To order an individual book, use the document-specific literature number:

TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals (literature number SPRU131) describes the TMS320C54x 16-bit, fixed-point, general-purpose digital signal processors. Covered are its architecture, internal register structure, data and program addressing, the instruction pipeline, DMA, and on-chip peripherals. Also includes development support information, parts lists, and design considerations for using the XDS510 emulator.

TMS320C54x DSP Reference Set, Volume 2: Mnemonic Instruction Set (literature number SPRU172) describes the TMS320C54x digital signal processor mnemonic instructions individually. Also includes a summary of instruction set classes and cycles.

TMS320C54x DSP Reference Set, Volume 3: Algebraic Instruction Set (literature number SPRU179) describes the TMS320C54x digital signal processor algebraic instructions individually. Also includes a summary of instruction set classes and cycles.

TMS320C54x DSP Reference Set, Volume 4: Applications Guide (literature number SPRU173) describes software and hardware applications for the TMS320C54x digital signal processor. Also includes development support information, parts lists, and design considerations for using the XDS510 emulator.

TMS320C54x Assembly Language Tools User's Guide (literature number SPRU102) 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 'C54x generation of devices.

TMS320C5xx C Source Debugger User's Guide (literature number SPRU099) tells you how to invoke the 'C54x emulator, EVM, 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. It also includes a tutorial that introduces basic debugger functionality.

TMS320C54x Code Generation Tools Getting Started Guide (literature number SPRU147) describes how to install the TMS320C54x assembly language tools and the C compiler for the 'C54x devices. The installation for MS-DOS™, OS/2™, SunOS™, Solaris™, and HP-UX™ 9.0x systems is covered.

Related Documentation

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

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

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

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

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.

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

Trademarks

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

MS-DOS is a registered trademark and Windows is a trademark of Microsoft Corp.

PC, PC-DOS, and OS/2 are trademarks of International Business Machines Corp.

Solaris and SunOS are trademarks of Sun Microsystems, Inc.

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

XDS510 is a trademark of Texas Instruments Incorporated.

UNIX is a registered trademark of UNIX Systems Laboratories, Inc.

If You Need Assistance . . .

❑ World-Wide Web Sites

TI Online	http://www.ti.com
Semiconductor Product Information Center (PIC)	http://www.ti.com/sc/docs/pic/home.htm
DSP Solutions	http://www.ti.com/dsps
320 Hotline On-line™	http://www.ti.com/sc/docs/dsps/support.htm

❑ North America, South America, Central America

Product Information Center (PIC)	(972) 644-5580		
TI Literature Response Center U.S.A.	(800) 477-8924		
Software Registration/Upgrades	(214) 638-0333	Fax: (214) 638-7742	
U.S.A. Factory Repair/Hardware Upgrades	(281) 274-2285		
U.S. Technical Training Organization	(972) 644-5580		
DSP Hotline	(281) 274-2320	Fax: (281) 274-2324	Email: dsph@ti.com
DSP Modem BBS	(281) 274-2323		

DSP Internet BBS via anonymous ftp to ftp://ftp.ti.com/pub/tms320bbs

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32	Email: epic@ti.com
Deutsch	+49 8161 80 33 11 or +33 1 30 70 11 68		
English	+33 1 30 70 11 65		
Francais	+33 1 30 70 11 64		
Italiano	+33 1 30 70 11 67		
EPIC Modem BBS	+33 1 30 70 11 99		
European Factory Repair	+33 4 93 22 25 40		
Europe Customer Training Helpline		Fax: +49 81 61 80 40 10	

❑ Asia-Pacific

Literature Response Center	+852 2 956 7288	Fax: +852 2 956 2200
Hong Kong DSP Hotline	+852 2 956 7268	Fax: +852 2 956 1002
Korea DSP Hotline	+82 2 551 2804	Fax: +82 2 551 2828
Korea DSP Modem BBS	+82 2 551 2914	
Singapore DSP Hotline		Fax: +65 390 7179
Taiwan DSP Hotline	+886 2 377 1450	Fax: +886 2 377 2718
Taiwan DSP Modem BBS	+886 2 376 2592	

Taiwan DSP Internet BBS via anonymous ftp to ftp://dsp.ee.tit.edu.tw/pub/TI/

❑ Japan

Product Information Center	+0120-81-0026 (in Japan)	Fax: +0120-81-0036 (in Japan)
	+03-3457-0972 or (INTL) 813-3457-0972	Fax: +03-3457-1259 or (INTL) 813-3457-1259
DSP Hotline	+03-3769-8735 or (INTL) 813-3769-8735	Fax: +03-3457-7071 or (INTL) 813-3457-7071
DSP BBS via Nifty-Serve	Type “Go TIASP”	

❑ Documentation

When making suggestions or reporting errors in documentation, please include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail: Texas Instruments Incorporated
Technical Documentation Services, MS 702
P.O. Box 1443
Houston, Texas 77251-1443

Email: comments@books.sc.ti.com

Note: When calling a Literature Response Center to order documentation, please specify the literature number of the book.

Contents

Figures

Tables

Examples

Chapter 1

Introduction

The TMS320C54x devices are high-performance CMOS microprocessors, optimized for digital signal processing applications.

The TMS320C54x DSPs are fully supported by a complete set of code generation tools, including an optimizing C compiler, assembler, linker, archiver, software simulator, full-speed emulator, and software development board.

The TMS320C54x devices are members of the TMS320 family of digital signal processors.

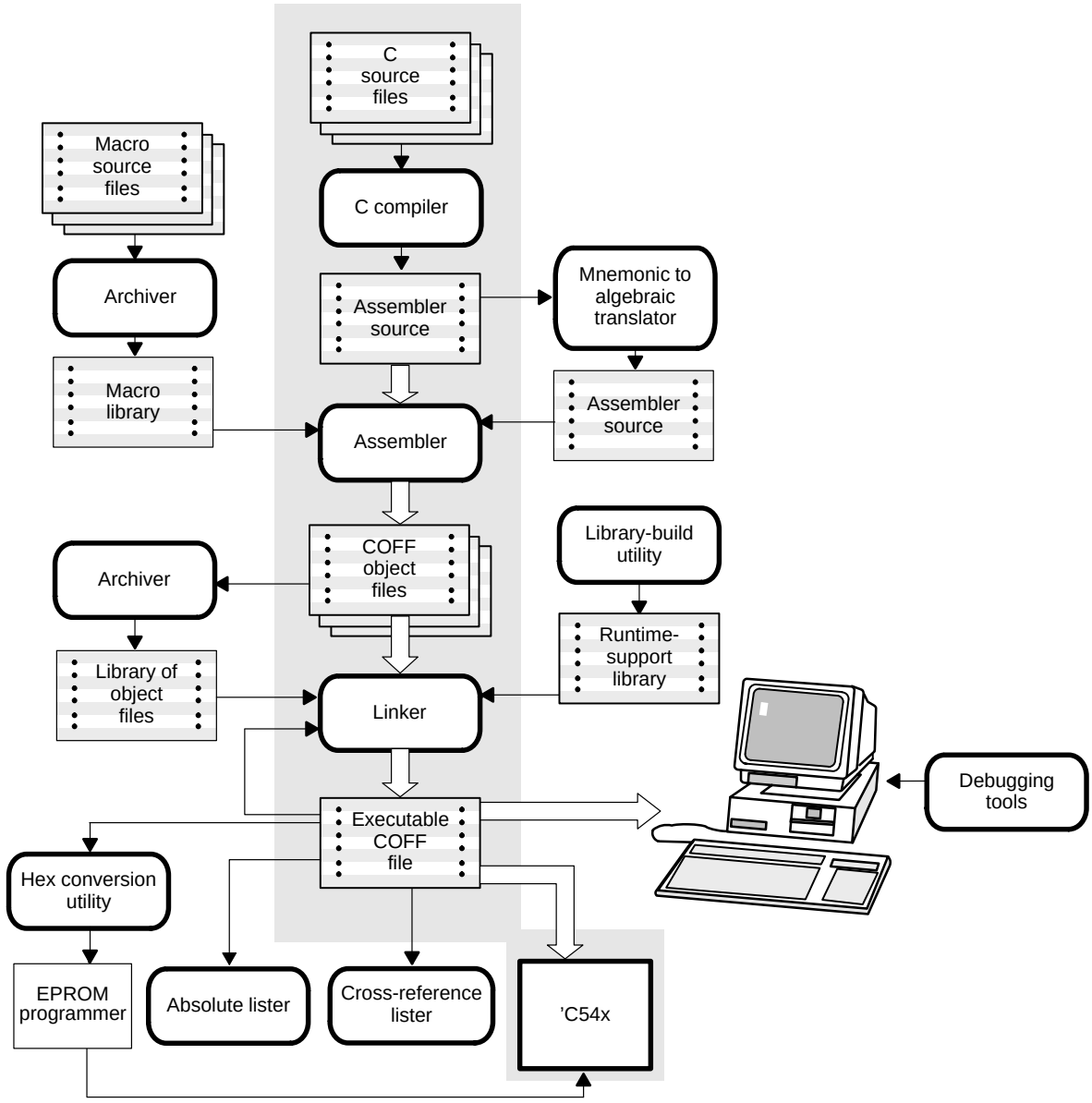
This chapter provides an overview of these tools and introduces the features of the optimizing C compiler. The assembler and linker are discussed in detail in the *TMS320C54x Assembly Language Tools User's Guide*.

Topic **Page**

1.1 Software Development Tools Overview

Figure 1–1 illustrates the ’C54x 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. TMS320C54x Software Development Flow



1.2 Tools Descriptions

- The **C compiler** translates C source code into 'C54x assembly language source code. A **shell program**, an **optimizer**, and an **interlist utility** are included in the compiler package.
  - The shell program enables you to automatically compile, assemble, and link source modules.
  - The optimizer modifies code to improve the efficiency of C programs.
  - The interlist utility incorporates C source statements with assembly language output.

  See Chapter 2, *Using the C Compiler*, for information about how to invoke the C compiler, the shell, the optimizer, and the interlist utility.
- The **assembler** translates assembly language source files into machine language object files. The machine language is based on common object file format (COFF). The *TMS320C54x Assembly Language Tools User's Guide* explains how to use the assembler.
- The **linker** combines object files into a single executable object module. As it creates the executable module, it adjusts references to symbols and resolves external references. The linker accepts relocatable COFF object files and object libraries as input. See Chapter 3, *Linking C Code*.
- The **archiver** collects a group of files into a single archive file, called a *library.* Additionally, the archiver allows you to modify a library by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is building a library of object modules (object library). An object library is shipped with the C compiler.
- The **mnemonic-to-algebraic translator utility** converts assembly language source files. The utility accepts an assembly language source file containing mnemonic instructions. It converts the mnemonic instructions to algebraic instructions, producing an assembly language source file containing algebraic instructions.
- The **library-build utility** builds your own customized, C, runtime-support library (see Chapter 7, *Library-Build Utility*). Standard runtime-support library functions are provided as source code in rts.src and as object code in rts.lib.

  The **runtime-support library** contains the ANSI standard runtime-support functions, compiler-utility functions, floating-point arithmetic functions, and C I/O functions that are supported by the 'C54x compiler (see Chapter 6, *Runtime-Support Functions*).

- The ’C54x debugger accepts executable COFF files as input, but most EPROM programmers do not. The **hex conversion utility** converts a COFF object file into TI-Tagged, ASCII-hex, Intel, Motorola-S, or Tektronix object format. You can download the converted file to an EPROM programmer. The *TMS320C54x Assembly Language Tools User’s Guide* explains how to use the hex conversion utility.

- The **absolute lister** accepts linked object files as input and creates .abs files as output. You assemble .abs files to produce lists that contain absolute rather than relative addresses. Without the absolute lister, producing such a listing would be tedious and require many manual operations. The *TMS320C54x Assembly Language Tools User’s Guide* explains how to use the absolute lister.

- The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definitions, and their references in the linked source files. The *TMS320C54x Assembly Language Tools User’s Guide* explains how to use the cross-reference utility.

The purpose of this development process is to produce a module that can be executed in a ’C54x target system. You can use one of several debugging tools to refine and correct your code. Available products include:

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

For information about these debugging tools, see the *TMS320C54x C Source Debugger User’s Guide*.

1.3 C Compiler Overview

The ’C54x C compiler is a full-featured optimizing compiler that translates standard ANSI C programs into ’C54x assembly language source. The following subsections describe key characteristics of the compiler.

1.3.1 ANSI Standard

The following features meet the ANSI standards for C:

- **ANSI standard C**

  The ’C54x compiler fully conforms to the ANSI standard for C as defined by ANSI 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. These extensions provide maximum portability and increased capability to your C code.

- **ANSI standard runtime support**

  The compiler package comes with a complete runtime library. All library functions conform to the ANSI standard for C. The library includes functions for string manipulation, dynamic memory allocation, data conversion, timekeeping, and trigonometry, plus exponential and hyperbolic functions. Functions for signal handling are not included, because these are target-system specific. For more information, see Chapter 6*, Runtime-Support Functions*.

1.3.2 Output Files

The following output files are created by the compiler:

- **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.

- **COFF object files**

  Common object file format (COFF) allows you to define your system’s memory map at link time. This maximizes performance by enabling you to link C code and data objects into specific memory areas. COFF also provides 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.

1.3.3 Compiler Interface

The following describe the compiler interface:

- **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, see Section 2.1, *About the Shell Program*, on page 2-2.

- **Flexible assembly language interface**

  The compiler has straightforward calling conventions, allowing you to easily write assembly and C functions that call each other. For more information, see Chapter 5, *Runtime Environment*.

1.3.4 Compiler Operation

The following describe the compiler operation:

- **Integrated preprocessor**

  The C preprocessor is integrated with the parser, which decreases compilation time. Standalone preprocessing or preprocessed listing is also available. For more information, see Section 2.2, *Controlling the Preprocessor*, on page 2-30.

- **Optimization**

  The compiler uses a sophisticated optimization pass that employs several advanced techniques for generating efficient, compact code from C source. The compiler performs general and ’C54x-specific optimizations. General optimizations can be applied to any C code. ’C54x-specific optimizations take advantage of the features unique to the ’C54x architecture. For more information about the C compiler’s optimization techniques, see Section 2.3, *Using the C Compiler Optimizer*, on page 2-36 and Appendix A, *Optimization*.

1.3.5 Utilities

The following describe the utilities accompanying the C compiler:

- **Source interlist utility**

  The compiler package includes a utility 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, see Section 2.5, *Using the Interlist Utility*, on page 2-45.

- **Library-build utility**

  The compiler package has a utility that creates a runtime-support library and installs standard header files for any configuration of compiler options that you choose. For more information, see Chapter 7, *Library-Build Utility*.

Chapter 2

Using the C Compiler

Translating your source program into code that the TMS320C54x can execute is a multistep process. You must compile, assemble, and link your source files to create an executable object file. The TMS320C54x package contains a special shell program, which 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 C compiler includes a utility that allows you to produce highly optimized code. The utility is explained in Section 2.3, *Using the C Compiler Optimizer*, on page 2-36.

The compiler package also includes a utility that includes your original C source statements as comments in the compiler's assembly language output. This enables you to inspect the assembly language code generated for each C statement. The utility is explained in Section 2.5, *Using the Interlist Utility*, on page 2-45.

Topic **Page**

2.1 About the Shell Program

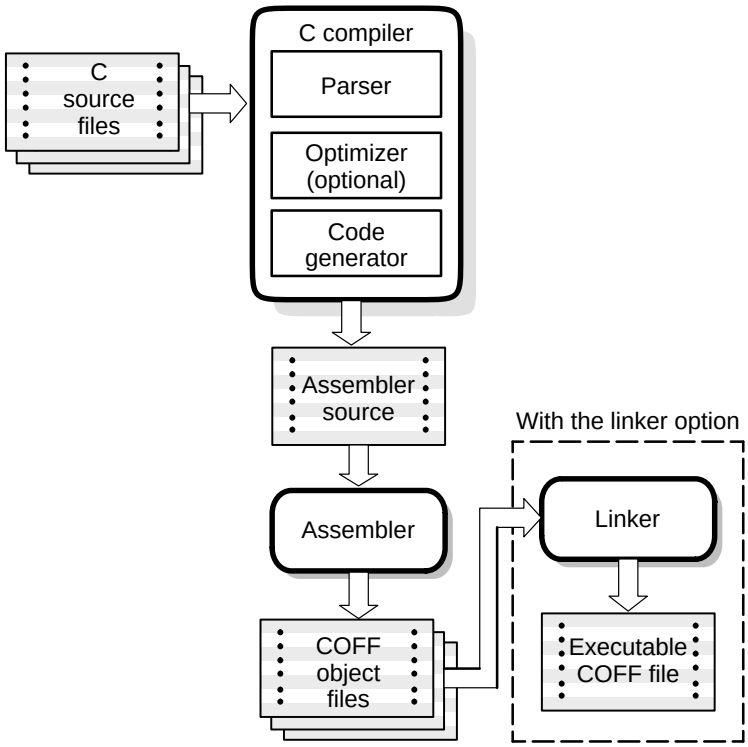
The shell program lets you compile, assemble, and optionally link in one step. The shell runs one or more source modules through the following:

- ❑ **Compiler** — The compiler parses and optimizes your source code to generate the assembler code.
- ❑ **Assembler** — The assembler generates a COFF object file.
- ❑ **Linker** (optional) — The linker links your object files to create an executable object file. The linker is optional at this point. You can compile various files with the shell and link later.

For more information about the 'C54x assembler and linker, refer to the *TMS320C54x Assembly Language Tools User's Guide* and Chapter 3, *Linking C Code*.

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

Figure 2–1. The Shell Program Overview



2.1.1 Invoking the C Compiler

To run the compiler, enter:

cl500 [–*options*] *filenames* [–z [*link_options*] [*object files*]]

cl500	is the command that invokes the compiler and the assembler.
–*options*	affects 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	affects the way the linker processes input files.
object files	names the object files that the compiler creates.

Options control the way 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, the –z option and its associated information must follow all filenames and compiler options on the command line.

Example 2–1 shows the command that compiles two files named symtab and file, assembles a third file named seek.asm, links all three files, and uses the quiet option, which suppresses progress messages.

Example 2–1. Invoking the Compiler

```
cl500 –q symtab file seek.asm –z
```

2.1.2 What Happens When the Compiler Is Invoked

As the compiler encounters each source file, it prints the filename in square brackets [for C files] or angle brackets <for asm files>. Example 2–1 uses the –q option to suppress the additional progress information that the compiler produces. Entering the command in Example 2–1 produces the output shown in Example 2–2.

Example 2–2. Suppressing Progress Messages

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

The normal progress information consists of a banner for each compiler pass and the names of functions as they are defined. Example 2–3 shows the output from compiling a single module *without* the –q option:

Example 2–3. Showing All Progress Messages

```
[symtab]
TMS320C54x ANSI C Compiler      Version x.xx
Copyright (c) 1997    Texas Instruments Incorporated
   ”symtab.c”   ==> main
   ”symtab.c”   ==> lookup
TMS320C54x ANSI C Codegen       Version x.xx
Copyright (c) 1997    Texas Instruments Incorporated
   ”symtab.c”:  ==> main
   ”symtab.c”:  ==> lookup
TMS320C54x COFF Assembler       Version x.xx
Copyright (c) 1997    Texas Instruments Incorporated
 PASS 1
 PASS 2

No Errors, No Warnings
```

2.1.3 Changing Compiler Behavior With Compiler Options

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

The following apply to the compiler options:

- Options are either single letters or two-letter pairs
- Options *are not* case sensitive
- Options are preceded by a hyphen
- Single-letter options without parameters can be combined. For example, –sgq is equivalent to –s –g –q.
- Two-letter pair options that have the same first letter can be combined. For example, –ms and –mb can be combined as –msb.
- Options that have parameters, such as –u*name* and –i*dir*, must be specified separately.
- Options with parameters can have a space between the option and parameter or be right next to each other.
- Files and options can occur in any order except the –z option. The –z option must be last and is followed by linker options.

You can define default options for the shell by using the C_OPTION environment variable. For a detailed description of the C_OPTION environment variable, see subsection 2.1.15, *Setting Options With the C_OPTION Environment Variable*, on page 2-28.

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

Table 2–1. Options Summary Table

(a) Options that control the compiler shell

Option	Effect	Page(s)
–@*filename*	Interprets contents of a file as an extension to the command line	2-14
–c	Disable linking (negates –z)	2-14, 3-6
–d*name*[=*def*]	Predefine a constant (or *name*)	2-14
–g	Enable symbolic debugging	2-14
–ga	Write a .global statement for all extern variables	2-14
–i*dir*	Define #include search path	2-14, 2-32
–k	Keep .asm file	2-14
–n	Create .asm file but don't run assembler	2-15
–q	Suppress progress messages (quiet)	2-15
–qq	Suppress all messages (super quiet)	2-15
–r*register*	Reserve global register	2-15
–s	Interlist optimizer comments (if available) with .asm source; otherwise, interlist C with .asm source	2-15, 2-45
–ss	Interlist C with .asm source	2-15, 2-45
–u*name*	Undefine *name*	2-15
–v*value*	Determines for which processor instructions are built	2-16
–z	Enable linking	2-16

Table 2–1. Options Summary Table (Continued)

(b) Options that change the file naming conventions

Option	Effect	Page(s)
–ea	Set default assembly file extension	2-17
–eo	Set default object file extension	2-17
–fa *file*	Identify assembly language file (default for .asm or .s*)	2-17
–fc *file*	Identify C source file (default for .c or no extension)	2-17
–fo *file*	Identify object file (default for .o*)	2-17
–fr *dir*	Specify object file directory	2-18
–fs *dir*	Specify assembly file directory	2-18
–ft *dir*	Override TMP environment variable	2-18

(c) Options that control definition-controlled inline function expansion

Option	Effect	Page(s)
–x0	Disable inline expansion of intrinsic operators	2-20, 2-41
–x1	Enable inline expansion of intrinsic operators (default)	2-20, 2-41
–x2 or –x	Define the symbol _INLINE and invoke the optimizer at level 2	2-20, 2-41

(d) Options that relax prototype type checking

Option	Effect	Page(s)
–tf	Relax prototype checking	2-21
–tp	Relax pointer combination checking	2-21

Table 2–1. Options Summary Table (Continued)

(e) Options that change the C runtime model

Option	Effect	Page(s)
–ma	Assume aliased variables	2-22
–mb	Disable RPT instruction	2-22
–mf	All calls are far calls and all returns will be far returns	2-22
–mn	Enable optimizer options disabled by –g	2-22
–mo	Disable back-end optimizer	2-22
–ms	Optimize for code space	2-22
–mx	Avoid ’C54x first run silicon bugs	2-22

Table 2–1. Options Summary Table (Continued)

(f) Options that control the parser

Option	Effect	Page(s)
–p?	Enable trigraph expansion	2-18
–pe	Treat code-E errors as warnings	2-18, 2-49
–pf	Generate function prototype listing file	2-18
–pk	Allow K&R compatibility	2-19, 4-18
–pl	Generate preprocessed listing (.pp file)	2-19, 2-35
–pm	Combines source files to perform program-level optimization	2-19
–pn	Suppress #line directives in .pp file	2-19
–po	Preprocess only	2-19
–pr	Generate error listing	2-20
–pw0	Disable all warning messages	2-20, 2-49
–pw1	Enable serious warning messages (default)	2-20, 2-49
–pw2	Enable all warning messages	2-20, 2-49
–px*filename*	Names the output file created when using the –pm option	2-20

Table 2–1. Options Summary Table (Continued)

(g) Options that control the optimizer

Option	Effect	Page(s)
–o0	Optimize at level 0 (register optimization)	2-22
–o1	Optimize at level 1 (level 0 plus local optimization)	2-22
–o2 or –o	Optimize at level 2 (level 1 plus global optimization)	2-22
–o3	Optimize at level 3 (level 2 plus file optimization)	2-22
–oi*size*	Set automatic inlining size (–o3 only)	2-22, 2-40
–ol0	File alters a standard library function	2-23
–ol1	File defines a standard library function	2-23
–ol2	File does not define or alter library functions	2-23
–on0	Disable optimizer information file	2-23
–on1	Produce optimizer information file	2-23
–on2	Produce a verbose optimizer information file	2-23
–op0	Functions in other files may call functions and modify variables defined here	2-24
–op1	Functions in other files do not call functions defined here but may modify variables defined here (default)	2-24
–op2	Functions in other files do not call functions or modify variables defined here	2-24
–op3	Functions in other files may call functions defined here, but may not modify variables defined here	2-24
–os	Interlist optimizer comments with .asm source	2-24, 2-45

Table 2–1. Options Summary Table (Continued)

(h) Options that control the assembler

Option	Effect	Page(s)
–aa	Allow absolute listing directives	2-25
–ad*symbol*	Define the symbol	2-25
–ahc*filename*	Copy the file	2-25
–ahi*filename*	Include the file	2-25
–al	Produce assembly language listing file	2-25
–as	Keep labels as symbols	2-25
–au*symbol*	Undefine the symbol	2-25
–ax	Produce cross-reference file	2-25

Table 2–1. Options Summary Table (Concluded)

(i) Options that control the linker

Option	Effect	Page(s)
–a	Generate absolute output (default)	2-26
–ar	Generate relocatable output module	2-26
–b	Disable merge of symbolic debugging data	2-26
–c	Use ROM initialization (default)	2-26
–cr	Use RAM initialization	2-26
–e*global_symbol*	Define entry point	2-26
–f*fill_value*	Define fill value	2-26
–g*global_symbol*	Keep C symbol global (overrides –h)	2-26
–h	Make all global symbols static	2-26
–heap*size*	Set heap size (bytes)	2-26
–i*dir*	Define library search path	2-26
–l*filename*	Supply library name	2-26
–m*filename*	Name the map file	2-26
–n	Ignore all fill specifications in MEMORY directives	2-26
–o*filename*	Name the executable output file	2-27
–q	Suppress progress messages (quiet)	2-27
–r	Keep relocation entries in output module	2-27
–s	Strip symbol table	2-27
–stack*size*	Set stack size (bytes)	2-27
–u*symbol*	Undefine symbol	2-27
–v*n*	Specify the output COFF format. The default format is COFF2.	2-27
–w	Displays a message when an undefined output section is created.	2-27
–x	Force rereading of libraries	2-27

2.1.4 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
.c or none (.c assumed)	C source
.asm, .abs, or .s* (extension begins with s)	Assembly source
.o* (extension begins with o)	Object

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, see subsection 2.1.6, *Options that Specify How the Shell Program Names Files*, on page 2-17.

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

You can use wildcard characters to compile or assemble multiple files. Wildcard specifications vary by system; use the appropriate form. For example, to compile all the files in a directory, enter the following:

```
cl500 *.c
```

2.1.5 Options That Control the Compiler Shell

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

–@*filename* appends the contents of a file to the command line. Use this option to avoid limitations on command line length imposed by the host operating system. Use a # or ; at the beginning of a line in the command file to include comments. Use /* to start and */ to end a comment at the end of a line.

–c suppresses the linking option, which prevents the shell from running 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 about the –c option, see subsection 3.2, *Invoking the Linker With the Compiler*, on page 3-6.

–d*name*[**=***def*] predefines the constant *name* for the preprocessor. It is equivalent to inserting the C statement #define *name def* at the top of each C source file. If the optional [*def*] is omitted, –d*name* sets *name* equal to 1.

–g causes the compiler to generate symbolic debugging directives for use with the C source-level debuggers.

–ga causes the compiler to generate a .global statement in the .asm file for each extern variable, even if the compiler thinks it is not referenced. This is useful if the only reference to an extern is contained within an asm statement. The compiler treats asm statements as comments and does not recognize any references within them. This option provides compatibility with previous versions of the compiler.

–i*dir* adds *dir* to the list of directories the compiler searches for included files. You can use this option a maximum of 10 times to define several directories; be sure to separate –i options with spaces. If you don't specify a directory name, the preprocessor ignores the –i option. For more information, see subsection 2.2.2, *The Search Paths for #include Files*, on page 2-32.

–k keeps the assembly language file. Normally, the shell deletes the output assembly language file after compilation is finished, 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.

–r*register* reserves *register* globally so that the code generator and optimizer cannot use it as a normal save-on-entry register. For information on creating global register variables, see Section 4.5, *Creating Global Register Variables*, on page 4-7.

–s interlists optimizer comments with the assembly language output of the compiler, if the optimizer has been specified; otherwise it invokes the interlist utility, which includes C source statements as comments in the assembly language output of the compiler. This option automatically uses the –k option. For more information, see Section 2.5, *Using the Interlist Utility*, on page 2-45.

–ss invokes the interlist utility, which includes C source statements as comments in the assembly language output of the compiler. This makes it easy for you to inspect the assembly code generated for each C statement. This option automatically uses the –k option. For more information, see Section 2.5, *Using the Interlist Utility*, on page 2-45.

–u*name* undefines the predefined constant *name*. Overrides any –d options for the specified constant.

–v*value* determines which processor instructions are built for. Use one of the following for *value*:

541
542
543
545
545lp
546lp
548

–z enables the linking option. This option causes the shell to run the linker on specified object files. The –z option must follow all source files and compiler options on the command line. All arguments that follow –z on the command line are passed to, and interpreted by, the linker. For more information, see Section 3.1, *Invoking the Linker as an Individual Program*, on page 3-2.

2.1.6 Options That Specify How the Shell Program Names Files

–ea overrides cl500's default naming conventions for filename
–eo extensions on assembly files and object files. The two –e options are listed below.

–ea [.]*asmext* for assembly files (default is .asm)

–eo [.]*objext* for object files (default is .obj)

For example, the following command assembles the file fit.rrr and creates an object file named fit.o37:

```
c1500 –ea .rrr –eo .o37 fit.rrr
```

The "." in the extensions and the space between the option and the extension are optional (the example could have been –earrr and –eoo37 instead of –ea .rrr and –eo .o37).

The –e option affects both the interpretation of filenames on the command line and the names of the output files and must always precede any filename on the command line.

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

–fa *file* for assembly language source file

–fc *file* for C source file

–fo *file* for object file

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

```
c1500 –fc cfile.s –fa asmbly
```

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

2.1.7 Options That Specify Directories

–fr *dir* permits you to specify a directory for object files. If the –fr option is not specified, the shell places 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:

```
cl500 –fr d:\object ...
```

–fs *dir* permits you to specify a directory for assembly files. If the –fs option is not specified, the shell places 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 *dir* permits you to specify a directory for temporary intermediate files. The –ft option overrides the TMP environment variable (described in subsection 2.1.16, *Setting a Temporary-File Directory With the TMP Environment Variable*, on page 2-29). To specify a temporary directory, insert the directory's pathname on the command line after the –ft option:

```
cl500 –ft d:\temp ...
```

2.1.8 Options That Control the Parser

–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, see the ANSI C specification, § 2.2.1.1, or K&R A12.1.

–pe causes the parser to treat code-E errors as warnings, allowing complete compilation. Normally, the code generator does not run if the parser detects any code-E errors. Note that the code-F errors are always fatal. For more information, see Section 2.6, *How the Compiler Handles Errors*, on page 2-48.

–**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. The –pf option is useful when conforming code to the ANSI C standard or generating a listing of procedures defined.

–pk relaxes certain requirements newly imposed by the ANSI C standard (that are stricter than those required by earlier K&R compilers). This facilitates compatibility between K&R C programs and the 'C54x ANSI compiler. The effects of the –pk option are described in Section 4.8, *Compatibility with K&R C*, on page 4-18.

–pl generates a preprocessed listing file. The compiler writes a modified version of the source file to an output file called file.pp. The .pp file contains all the source from #include files and expanded macros; it does not contain any comments or code for false #if or #ifdef 'C54x compiler directives. For more information, see subsection 2.2.3, *Generating a Preprocessed Listing File*, on page 2-35.

–pm when used with the –o3 option, combines source files into one intermediate file called a module to perform program-level optimization instead of file-level optimization. The module proceeds to the optimization and code generation passes of the compiler. Because the compiler can now see the entire C program, it performs several optimizations that are not usually done during file-level optimization:

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

–pn suppresses line and file information. The –pn option suppresses the #line 'C54x compiler directives of the following form in the .pp file generated with –po or –pl.

```
#line 123 "file.c"
```

The –pn option is useful when you are compiling machine-generated source.

–po runs the compiler for preprocessing only. When invoked with the –po option, the compiler processes only macro expansions, included files, and conditional compilation. The compiler writes the preprocessed file with a .pp extension. For more information, see subsection 2.2.3, *Generating a Preprocessed Listing File*, on page 2-35.

–pr creates a parser error message file. The error file has the base name of the input file and an .err extension. The file contains all error messages generated by the parser during compilation fo the specified file.

–pw0 disables all warning messages. This is useful when you are aware of the condition causing the warning and consider it innocuous. For more information, see Section 2.6, *How the Compiler Handles Errors*, on page 2-48.

–pw1 enables serious warning messages. This is the default. For more information, see Section 2.6, *How the Compiler Handles Errors*, on page 2-48.

–pw2 enables all warning messages. For more information, see Section 2.6, *How the Compiler Handles Errors*, on page 2-48.

–px*filename* when using the –pm option, specifies the name of the final output file.

2.1.9 Options That Control Definition-Controlled Inline Function Expansion

For more information on –x options, see subsection 2.4.2.2, *Defining and Using the _INLINE Preprocessor Symbol*, on page 2-42.

–x0 causes no inline expansion. This option 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 or **–x** creates the preprocessor symbol _INLINE, assigns it the value 1, and invokes the optimizer at level 2, thereby enabling definition-controlled inline expansion.

2.1.10 Options That Relax Prototype Type Checking

–tf relaxes type checking on redeclarations of functions defined with a prototype. 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 promotions (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 types with different qualifications can be combined:

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

The –tp option 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.

2.1.11 Options That Change the C Runtime Model

–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 disables the uninterruptible RPT instruction.

–mf interprets all call instructions as far calls, and interprets all return instructions as far returns. A far call calls a function outside the 16-bit range, and a far return returns a value from outside the 16-bit range.

–mn reenables 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.

–mo disables the back-end optimizer.

–ms optimizes for code space instead of for speed.

–mx supports first-pass TMX silicon. The –mx option enables the code generator to work around some of the known hardware bugs in early ’C54x devices.

2.1.12 Options That Control the Optimizer

–o*n* causes the compiler to optimize the intermediate file that is produced by the parser. The *n* denotes the level of optimization. There are four 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, see Section 2.3, *Using the C Compiler Optimizer*, on page 2-36 and Appendix A, *Optimization*. For more information on the –o*n* options, see subsection 2.3.1, *Optimization Levels*, on page 2-37.

–oi*size* controls automatic inlining of functions (not defined or declared as *inline*) at optimization level 3 (–o3). You specify the *size* limit for the largest function that will be inlined. If the –oi option is not used, the optimizer will inline very small functions when invoked at level 3. Setting the size to 0 (–oi0) disables automatic inlining completely. Note that the –x option controls inlining of functions declared with the *inline* keyword (see subsection 2.1.9, *Options That Control Definition-Controlled Inline Function Expansion*, on page 2-20).

–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 the file you are compiling redefines any of these standard functions, the compiler may produce incorrect code. Use the –ol option to notify the optimizer if any of the following situations exist:

- **–ol0**: This file defines a function with the same name as a standard library function.
- **–ol1**: This file contains the standard library definition functions for those functions that are defined in it.
- **–ol2**: The file does not alter standard library functions. Use this option to restore the default behavior of the optimizer if you have used one of the other two –ol options in a command file, an environment variable, and so on.

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

- **–on0**: Do not produce an information file. Use this option to restore the default behavior of the optimizer if you have used one of the other two –on options in a command file, an environment variable, and so on.
- **–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 external functions or modify this file's external variables. Level 3 optimization (–o3) combines this information with its own file-level analysis to decide whether to treat this file's external function and variable definitions as if they had been declared STATIC. The following three levels are defined:

- **–op0**: Signals the optimizer that functions in other modules might call functions or variables defined in the current module. This disables some of the –o3 optimizations.
- **–op1**: Signals the optimizer that no functions exist in other modules that might call functions defined in this module, and that no interrupt function defined elsewhere might call functions defined here. This is the default when –o3 is used.
- **–op2**: 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.
- **–op3:** Signals the optimizer that functions in this module may be called by other modules, but no variables declared within the module may be altered by other modules. This disables some of the –o3 optimizations.

Level –op1 reverts to –op0 if the file does not define the function *main*, does not define an interrupt function, or if it contains calls to unknown functions. Level –op0 is the default since it is unlikely that an interrupt function defined elsewhere would call a user function that is defined in a file containing main (and all of the user functions called directly or indirectly from main).

Level –op2 also reverts to –op0 if no main or interrupt functions are found, but unlike level –op1, it does not revert if there are calls to unknown functions.

Use of the –op options automatically invokes the optimizer at level 3 (–o3).

–os interlists optimizer comments into the compiler's assembly language output. For more information, see subsection 2.5.2 on page 2-46.

2.1.13 Options That Control the Assembler

For more information about assembler options, see the *TMS320C54x Assembly Language Tools User's Guide.*

–aa tells the assembler to use its –a option to allow absolute listing directives in the input file. The assembler does not produce an object file when this option is used.

–ad*symbol* tells the assembler to define the specified symbol for the assembly module.

–ahc*filename* tells the assembler to copy the specified file for the assembly module. The file is inserted before source file statements. The copied file appears in the assembly listing files.

–ahi*filename* tells the assembler to include the specified file for the assembly module. The file is included before source file statements. The included file does not appear in the assembly listing files.

–al tells the assembler to use its –l (lowercase L) option to produce an assembly language listing file.

–as tells the assembler to use its –s option to retain labels. Label definitions are written to the COFF symbol table for use with symbolic debugging.

–au*symbol* tells the assembler to undefine the specified symbol for the assembly module.

–ax tells the assembler to use its –x option to include cross-reference information in the listing file.

2.1.14 Options That Control the Linker

All command-line input following –z is passed to the linker as parameters and options. For more information about linker options, see the *TMS320C54x Assembly Language Tools User's Guide*.

–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 auto-initialization model of the 'C54x C compiler.

–cr enables linking conventions defined by the RAM auto-initialization model of the 'C54x 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.

–g*global_symbol* keeps C *global_symbol* global. The –g option overrides the –h option for the symbol.

–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. The default size is 1K bytes.

–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.

–n forces the linker to ignore any MEMORY directive fill specification.

–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*register*	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. The default size is 1K words.
–u*symbol*	places the unresolved external *symbol* into the output module's symbol table.
–v*n*	specifies the output COFF format. The default format is COFF2.
–w	displays a message when an undefined output section is created.
–x	forces rereading of libraries and resolves back references.

2.1.15 Setting Options With 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, 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 with the same syntax 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; set up the C_OPTION environment variable as follows:

Operating System	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:

```
cl500 *.c              ; compiles and links
cl500 -c *.c           ; only compiles
cl500 *.c -z c.cmd     ; compiles/links using command file
cl500 -c *.c -z c.cmd  ; only compiles (-c overrides -z)
```

2.1.16 Setting a Temporary-File Directory With the TMP Environment Variable

The shell program creates temporary, intermediate files as it processes your program. For example, the parser phase of the shell creates an intermediate 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 intermediate 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:

- Use the TMP environment variable:

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

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

- Use the –ft option on the command line:

  ```
  cl500 –ft d:\temp....
  ```

  The –ft option overrides the TMP environment variable (for more about the –ft option, see subsection 2.1.7, *Options That Specify Directories*, on page 2-18).

2.2 Controlling the Preprocessor

The ’C54x C compiler includes standard C preprocessing functions, which are built into the first pass of the compiler. The preprocessor resolves the following:

- Macro definitions and expansions
- #include files
- Conditional compilation
- Various other preprocessor directives. Directives instruct the preprocessor to perform specific actions. Directives are specified in the source file by lines beginning with the # character. For more information about preprocessor directives, see subsections 2.2.3, *Generating a Preprocessed Listing File*, on page 2-35 and 2.2.4, *Creating Custom Error Messages*, on page 2-35.

This section describes specific features of the ’C54x 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*.
_TMS320C5xx	Expands to 1 (identifies the ’C54x processor).
_INLINE	Expands to 1 under the –x or –x2 optimizer option, undefined otherwise.
C_MODE	Specifies that all calls and branches are within the normal 16-bit address range (default operation).
.FAR_MODE	Specifies that all calls and branches are to an extended address space (used for the extended addressing ability of the ’C548).

† 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” , ”13:58:17”, ”Jan 14 1994”);
```

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

2.2.2 The Search Paths for #include Files

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, partial path information, or a filename with no path information.

If you enclose the filename in *double quotes*, the compiler searches for the file in the following directories, in the order 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) Directories named with the –i compiler option in the shell

3) Directories set with the C_DIR environment variable

If you enclose the filename in *angle brackets*, the compiler searches for the file in the following directories, in the order given:

1) Directories named with the –i compiler option in the shell
2) Directories set with the C_DIR environment variable

Note: Enclosing the Filename in <angle brackets>

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

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

2.2.2.1 Changing the #include File Search Path

The –i shell option names an alternate directory that contains #include files. The –i option has the following form:

```
cl500 –i pathname ...
```

You can use up to ten –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. The source.c file contains one of the following directive statements:

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

or

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

Assume that the complete pathname for alt.h is:

DOS or OS/2 c:\320tools\files\alt.h

UNIX /320tools/files/alt.h

The table below shows how to invoke the compiler. Select the row for your host system:

Operating System	Enter
DOS or OS/2	**cl500 –ic:\320tools\files source.c**
UNIX	**cl500 –i/320tools/files source.c**

2.2.2.2 C_DIR Environment Variable

The compiler uses the C_DIR environment variable 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:

Operating System	Enter
DOS or OS/2	`set C_DIR=c:\320tools\files`
UNIX	`setenv C_DIR "/320tools/files"`

Then you can include alt.h:

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

or

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

and invoke the compiler without the –i option:

```
cl500 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:

Operating System	Enter
DOS or OS/2	`set C_DIR=`
UNIX	`unsetenv C_DIR`

2.2.3 Generating a Preprocessed Listing File

The –pl shell option allows you to generate a preprocessed version of your source file. The compiler's preprocessing functions perform the following 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 (see subsection B.2.1, *Parsing in Two Passes*, on page B-6). The resulting preprocessed listing file is a valid C source file that can be rerun through the compiler.

2.2.4 Creating Custom Error Messages

The standard #error preprocessor directive forces the compiler to issue a diagnostic message and halt compilation. The 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 that of #error. For more information, see Section A12.7 of K&R.

2.3 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 performing such tasks as simplifying loops, rearranging statements and expressions, and allocating variables into registers.

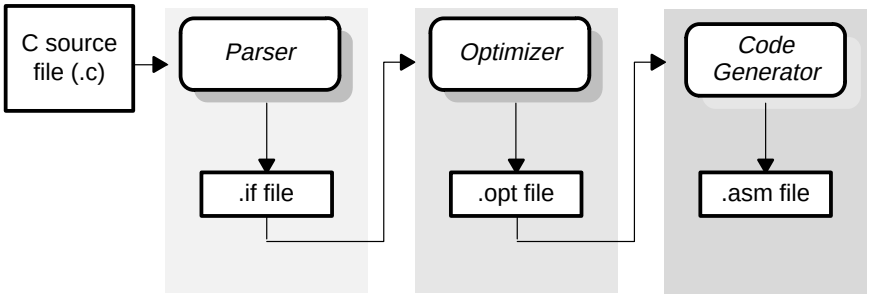
The optimizer runs as a separate pass between the parser and the code generator. The easiest way to invoke the optimizer is to use the cl500 shell program, specifying the –o option on the cl500 command line (you may also invoke the optimizer outside cl500; see Section B.3, *Invoking the Optimizer Individually*, on page B-7 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 optimizer defaults to level 2. The –o option may also be followed by the –ol or –on options with modifiers. For more information, see subsection 2.1.12, *Options That Control the Optimizer*, on page 2-22.

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

```
cl500 -o -x2 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 cl500 options –x, –os, –ma, –q, and –pk; these options are listed in Table 2–1 beginning on page 2-6.

2.3.1 Optimization Levels

There are four levels of optimization: 0, 1, 2, and 3. These levels control the type and degree of optimization, as described in the following list:

- **Level 0**
  - Performs control-flow-graph simplification
  - Allocates variables to registers
  - Performs loop rotation
  - Eliminates dead code
  - Simplifies expressions and statements
- **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 dead assignments
  - 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
  - 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 subsection 2.1.12, *Options That Control the Optimizer*, on page 2-22.)

The preceding list describes optimizations performed by the standalone optimization pass. The code generator performs several additional optimizations, particularly TMS320C54x-specific optimizations; it does so regardless of whether you invoke the optimizer. These optimizations are always enabled and are not affected by the optimization level you choose.

For more information about the meaning and effect of specific optimizations, see Appendix A, *Optimization*.

2.3.2 Debugging Optimized Code

When debugging a program, ideally you should debug it in an unoptimized form and verify 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 compiler 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 compiler option; –mn re-enables the optimizations disabled by –g.

2.3.3 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 ensure that your program performs as you intend.

2.3.3.1 Use Caution With asm Statements in Optimized Code

You must be extremely careful when using asm (inline assembly) statements in optimized code. The optimizer rearranges code segments, uses registers freely, and may completely remove variables or expressions. Although the compiler never optimizes out an asm statement (except when it is totally unreachable), the surrounding environment in which the assembly code is inserted may differ significantly from the C 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.

2.3.3.2 Use the Volatile Keyword for Memory Necessary Memory Accesses

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

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

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

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

```
volatile unsigned int *ctrl
```

2.3.3.3 Use Caution When Accessing Aliased Variables

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

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

2.4 Function Inlining

When an inline function is called, 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 (the code necessary to enter and exit the function) of a function call.
- Once inlined, the optimizer is free to make the function code as efficient as possible in the context of the surrounding code.

The compiler automatically expands the intrinsic operators of the target system (such as abs) by default. This happens whether or not you use the optimizer and whether or not you use any compiler or optimizer options on the command line. (You can defeat this automatic inlining by invoking the compiler with the –x0 option.) Functions that expand to intrinsic operators are:

- abs
- labs
- fabs

In addition, when you invoke the compiler with optimization, the compiler performs two other types of inline function expansion: automatic and definition controlled.

2.4.1 Automatic Inline Expansion

Automatic inline expansion is done when you invoke the compiler with level 3 optimization (–o3). By default, with this level of optimization, the compiler inlines very small functions automatically. You can change the size of functions that are automatically inlined with the –**oi***size* option. The –oi*size* option specifies that functions whose size is less than *size* units are inlined regardless of how they were declared. The compiler 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 in the next subsection). You can defeat all automatic inlining by setting the size to 0 (–oi0).

- If you set the size parameter to 0 (–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 compiler inlines all functions whose size is less than the *size* parameter. If the function is called more than once, the compiler multiplies the size of the function by the number of calls, and inlines the function only if the resulting product is less than the *size* parameter. The compiler measures the size of a function in arbitrary units. The optimizer information file (created with the –on1 or –on2 option) reports the size of each function in the same units that the –oi option uses.

2.4.2 Definition-Controlled Inline Function Expansion

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

- 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 is inlined only within the module where it is defined as inline. Since the function is not visible within other modules, the function is called normally from other modules, rather than inlined.
- By *declaring* a function as static inline (typically in a header file), you can specify that the function is inlined in any module where it is visible (typically, in any module that includes the header). When you do this, you will probably want to use the –x option and the _INLINE keyword to control its declaration. If you fail to do this and subsequently compile *without* the optimizer, the call to the function will be unresolved.

2.4.2.1 Using the Inline Keyword to Define or Declare Functions That Will Be Inlined

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

You can use the inline keyword in the following ways:

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

creates a *definition* of the function so that a global symbol for the function is defined and code for the function is generated. It also specifies that all calls to the function in the current source module will be inlined.

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

specifies that the function is to be expanded inline and that no code is generated for the function declaration itself. This usage is a *declaration* of the function, and **not** a definition. You may place functions declared in this way in header files and include them in all source modules of the program. Use the –x option and the _INLINE preprocessor symbol to control its declaration.

2.4.2.2 Defining and Using the _INLINE Preprocessor Symbol

A command line switch controls the definition of the _INLINE keyword and some types of inline function expansion.

–x0 no definition controlled inline expansion. Defeats the default expansions of the intrinsic operator functions. This option does not defeat the inline function expansions described in subsection 2.4.1, *Automatic Inline Expansion*, on page 2-40.

–x1 resets the default behavior. You use this option primarily to reset the default behavior from the command line if you have used another –x option in an environment variable or command file.

–x2 or **–x** creates the preprocessor symbol _INLINE, assigns it the value 1, and, if the optimizer was not invoked with a separate command line option, invokes the optimizer at the default level (–o2).

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

Example 2–4 on page 2-43 illustrates how the runtime-support library uses the _INLINE symbol.

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 used to conditionally define _ _INLINE so that strlen is declared as static inline only if the _INLINE symbol is defined.

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

Use the _INLINE preprocessor symbol in the same way that the function libraries use it, so that your programs run regardless of whether inlining mode is selected for any or all of the modules in your program.

Example 2–4. How the Runtime Support Library Uses the _INLINE Symbol

```
/*****************************************************************************/
/* string.h  v0.00                                                           */
/*****************************************************************************/
#ifndef _STRING
#define _STRING

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

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

#if _INLINE

__INLINE void *memcpy(void *s1, const void *s2, register size_t n)
{
    if (n)
    {
      register char *src  = s2;
      register char *dest = s1;
      do
     *dest++ = *src++;
      while (--n);
    }
    return s1;
}
#endif /* _INLINE */

#undef __INLINE
/*****************************************************************************/
/*  memcpy.c v0.00                                                           */
/*****************************************************************************/
#undef _INLINE
#include <string.h>

void *memcpy(void *s1, const void *s2, register size_t n)
{
    if (n)
    {
      register char *src  = s2;
      register char *dest = s1;

      do
       *dest++ = *src++;
      while (--n);
    }

    return s1;
}
```

There are two definitions of the memcpy function used in Example 2–4. The first, in the string.h header, is an inline definition. Note that this definition is enabled and the prototype is 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 the memcpy.c section, 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 string.h is included so that the noninline 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 are inlined and the linker does have to use the memcpy in the runtime-support library to resolve any references. Any modules that call memcpy and are not compiled with inlining enabled generate calls that the linker resolves by getting the memcpy code out of the library.

2.5 Using the Interlist Utility

The compiler package includes a utility that includes C source statements as comments in the assembly language output of the compiler. The interlist utility enables you to inspect the assembly code generated for each C statement. The interlist utility behaves differently depending on whether or not the optimizer is being used.

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

```
cl500 –ss function
```

2.5.1 Using the Interlist Utility Without the Optimizer

The interlist utility runs as a separate pass between the code generator and the assembler. It reads both the assembly and C source files, merges them, and writes the C statements into the assembly file as comments beginning with dashes, such as `;----`. The output assembly file, function.asm, is assembled normally. The –ss option automatically prevents the shell from deleting the interlisted assembly language file.

Example 2–5 shows a typical interlisted assembly file.

Example 2–5. An Interlisted Assembly Language File

```
   .state32
;  ac500 binsearch binsearch.if
   .global   _binsearch
;------------------------------------------------------------
;      int binsearch(int x, int v[], int n)
;------------------------------------------------------------
;*************************************************************
;* FUNCTION DEF: _binsearch                                  *
;*************************************************************
_binsearch:
         SUB       SP, SP, #24
;* A1    assigned to _x
;* A2    assigned to _v
;* A3    assigned to _n
;------------------------------------------------------------
;      int low, mid, high;
;------------------------------------------------------------
         STR       A3, [SP, #8]
         STR       A2, [SP, #4]
         STR       A1, [SP, #0]
;------------------------------------------------------------
;      low = 0;
;------------------------------------------------------------
         MOV       A1, #0
         STR       A1, [SP, #12]
;------------------------------------------------------------
;      high = n - 1;
;------------------------------------------------------------
         LDR       A1, [SP, #8]
         SUB       A1, A1, #1
         STR       A1, [SP, #20]
```

2.5.2 Using the Interlist Utility With the Optimizer

If the –os option is used with the optimizer (–o), the interlist utility does **not** run as a separate pass. Instead, the optimizer inserts comments into the code indicating how the optimizer has rearranged and optimized the code. These comments appear in the assembly language file as comments starting with `;**`.

Example 2–6 shows the function from Example 2–5 compiled with the optimizer (–o2) and –os.

Example 2–6. The Function From Example 2–5 Optimized

```
    .state32
;   armopt –s –O2 –e –Z binsearch.if binsearch.opt
    .global   _binsearch
;*************************************************************
;* FUNCTION DEF: _binsearch                                  *
;*************************************************************
_binsearch:
          STMFD     SP!, {V1,V2,LR}
;** 5  ----------------------    high = n–1;
          SUB       A4, A3, #1
;** 6  ----------------------    if ( n <= 0 ) goto g9;
          CMP       A3, #0
          BLE       L9
;** 4  ----------------------    low = 0;
          MOV       V1, #0
L3:
;**----------------------g3:
;** 8  ----------------------    mid = (low+high)/2;
          ADD       A3, A4, V1
          ADD       A3, A3, A3, LSR #31
          MOV       A3, A3, ASR #1
;** 9  ----------------------    C$1 = v[mid];
          LDR       V2, [A2, +A3, LSL #2]
;** 9  ----------------------    if ( x < C$1 ) goto g7;
          CMP       A1, V2
          BLT       L7
```

If the –os option is specified and the optimizer (–o) is not, the option will be ignored and the following warning will be issued:

```
Optimizer comments requested, optimization not run, ignoring option –os
```

If the –ss option is used with the optimizer (–o), the original C source code is added to the assembly file by the interlist utility. In this case, to maintain a correspondence between the interlist comments and the code, optimization and instruction scheduling will take place on a line by line basis, roughly corresponding to the instructions between interlist comments. If both the –ss and –os options are used, the file generated contains the assembly code, the original C source comments, and the optimized source comments.

The –s option will interlist optimizer comments into the assembly file if the comments are available; otherwise, it will use the interlist utility to interlist C source into the assembly file.

2.6 How the Compiler Handles Errors

An important function of the compiler 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 in C code are divided into four classes according to severity; these classes are identified by the letters *W*, *E*, *F*, and *I (upper-case i)*. The compiler also reports other errors that are not related to C but prevent compilation.

- **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 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. See subsection 2.6.1, *Treating Code-E Errors as Warnings*, on page 2-49 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 errors** are always 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 errors** are usually fatal: they result from 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 4.9, *Compiler Limits*, on page 4-20.) This is an example of a code-I error:

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

- **Other error messages** are usually fatal: they result from incorrect command line syntax or inability to find specified files.These errors are identified by the symbol >> preceding the message. This is an example of such an error:

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

2.6.1 Treating Code-E Errors as Warnings

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 generates 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.

2.6.2 Altering the Level of Warning Messages

You can determine which levels of warning messages to display by setting the warning message level with the –pw option. The number following –pw denotes the level (0,1, or 2). Use Table 2–3 to select the appropriate level. See subsection 2.6.3, *An Example of How You Can Use Error Options*, on page 2-50 for an example of the –pw option.

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

If you want to…	**Use option**
Disable all warning messages. This level is useful when you are aware of the condition causing the warning and consider it innocuous.	–pw0
Enable serious warning messages. This is the default.	–pw1
Enable all warning messages	–pw2

2.6.3 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 the following code segment:

```
func(char *pc)
{
   int *pi;

   pi = pc;
}
```

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

  ```
  "err.c", line 5: [E120] 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 –q and –pe options, this is the result:

  ```
  "err.c", line 5: [E120] 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 the –q, –pe, and –pw options, this is the result:

  ```
  [err.c]
  ```

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

Chapter 3

Linking C Code

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

- You can compile individual modules and then link them together. This is especially useful when you have multiple source files.
- You can compile and link in one step by using the compiler shell (cl500). This is useful when you have a single source module.

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

For a complete description of the linker, see the *TMS320C54x Assembly Language Tools User's Guide*.

Topic **Page**

3.1 Invoking the Linker as an Individual Program

The examples in this subsection show how to invoke the linker in a separate step after you have compiled and assembled your programs.

This is the general syntax for linking C programs in a separate step:

lnk500 {**–c** | **–cr**} *filenames* [*additional-options*] [**–o** *name.out*] **–l** *libraryname*

lnk500 is the command that invokes the linker.

–c | **–cr** are options that tell the linker to use special conventions defined by the C environment. When you use lnk500, you must use –c or –cr. The –c linker option uses the ROM model of initialization. The –cr linker option uses the RAM model of initialization. (See subsection 3.1.1, *About the –c and –cr Linker Options*, for more information.)

filenames are object files created by compiling C programs or assembling assembly language programs.

additional-options are options that control the linking process (see subsection 3.1.4, *Additional Linker Options*, on page 3-4).

–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 *libraryname* 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.

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

```
lnk500 –c prog1 prog2 prog3 –l rts.lib –o
prog.out
```

The rts.lib library is included with the C compiler. For information about the rts.lib library and its contents, see Section 6.1, *About the Runtime–Support Libraries*, on page 6-2.

3.1.1 About the –c and –cr Linker Options

The C compiler produces tables of data for autoinitializing global variables. Subsection 5.8.2, *Autoinitialization of Variables and Constants*, on page 5-27 discusses the format of these tables. 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 that are initialized at *load time* use the –cr linker option. For more information about the RAM model, see subsection 5.8.2.1, *Initializing Variables in the RAM Model*, on page 5-28.

- **ROM Model** (–c linker option)

  Global variables that are initialized at *runtime* use the –c linker option. For more information about the ROM model, see subsection 5.8.2.2, *Initializing Variables in the ROM Model*, on page 5-29.

3.1.2 Linking Conventions for the –c and –cr Linker Options

When 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. That is, they tell the linker to use the ROM or RAM model of autoinitialization. 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. The _c_int00 symbol is automatically referenced; this ensures that boot.obj is automatically linked in from the runtime-support library, rts.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 following occur:

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

  - The STYP_COPY flag (0010h) 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.
- The **.const** section is placed on the data page (page 1).

3.1.3 Libraries and Allocation

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

```
lnk500 –c prog1 prog2 prog3 –l rts.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. These directives are described in the *TMS320C54x Assembly Language Tools User's Guide*.

3.1.4 Additional Linker Options

All command-line input following the –z option is passed to the linker as parameters and options. Following are the options that control the linker with detailed descriptions of their effects. For more information on linker options, see the linker description in the *TMS320C54x Assembly Language Tools User's Guide*.

Option	Effect
–a	Generate absolute output (default)
–ar	Generate relocatable output module
–b	Disable merge of symbolic debugging data
–e*global_symbol*	Define entry point
–f*fill_value*	Define fill value
–g*global_symbol*	Keep C symbol global (overrides –h)
–h	Make all global symbols static
–heap *size*	Set heap size (bytes); see subsection 3.1.5, *Sizing the Stack and Heap*, on page 3-5 for more information
–i*dir*	Define library search path
–k	Ignore alignment flags in input sections
–m*filename*	Name the map file

Option	Effect
–n	Ignore all fill specifications in MEMORY directives
–q	Suppress progress messages (quiet)
–r	Keep relocation entries in output module
–s	Strip symbol table
–stack*size*	Set stack size (bytes); see subsection 3.1.5, *Sizing the Stack and Heap*, on page 3-5 for more information
–u*symbol*	Undefine symbol
–w	Display a message when an undefined output section is created.
–x	Force rereading of libraries

3.1.5 Sizing the Stack and Heap

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

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

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

The linker always includes the .stack section; it 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 0, so any –stack or –heap option takes effect). The default size for the .stack section is 1K words; the default for the .sysmem section is 1K words.

3.2 Invoking the Linker With the Compiler

The options and parameters discussed in this section apply to both methods of linking; however, when you link using 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 (for more information about C_OPTION, see subsection 2.1.15, *Setting Options With the C_OPTION Environment Variable*, on page 2-28).
- –z divides the command line into compiler options (before –z) and linker options (following –z)
- –c suppresses –z, so do not use –c if you want to link

For all arguments that follow –z on the command line, the shell generates a linker command file containing the arguments and passes the command file to the linker. The arguments can be other linker command files, object files, linker options, or libraries. For example, to compile and link all the .c files in a directory, enter:

```
cl500 –sq *.c –z lnk.cmd –o prog.out –l rts.lib
```

First, the shell compiles all files with the .c extension by using the –s and –q options. Second, because –z is specified, the linker links the resulting object files by using the generated linker command file.

The order in which the linker processes arguments can be important, especially for command files and libraries. When you use the shell to run the linker, it 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

You can override the –z option by using the –c shell option. This 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 –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 the RAM model of initialization, use the –cr linker option.

3.3 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 *TMS320C54x Assembly Language Tools User's Guide.*

3.3.1 Linking With 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:

lnk500 {**–c** | **–cr**} *filenames* **–l** *libraryname*

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

All C programs must be linked with an object module called *boot.obj.* When a program begins running, it executes boot.obj first. The boot.obj module 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. (See subsection 3.1.1, *About the –c and –cr Linker Options*, on page 3-3 for more information.)

Chapter 6, *Runtime-Support Functions*, describes additional runtime-support functions that are included in rts.lib. These functions include ANSI C standard runtime support.

3.3.2 Sections Created by the Compiler

The compiler produces seven 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 3–1 summarizes the sections.

Table 3–1. Sections Created by the Compiler

Name	Type	Contents
.text	Initialized	Executable code
.cinit	Initialized	Tables for explicitly initialized global and static variables
.const	Initialized	String literals and global and static const variables that are explicitly initialized
.switch	Initialized	Switch statement tables
.bss	Uninitialized	Global and static variables
.stack	Uninitialized	Software stack area
.sysmem	Uninitialized	Dynamic memory area for malloc functions

When you link your program, you must specify where to locate the sections in memory and the amount of memory to allocate for each. In general, initialized sections can be linked into ROM or RAM; uninitialized sections must be linked into RAM. Although you can place the .const section in ROM because it is never written to, it must be configured as data memory because of how it is accessed. The linker provides MEMORY and SECTIONS directives for allocating sections. See Section 5.1, *Memory Model*, on page 5-2 and the *TMS320C54x Assembly Language Tools User's Guide* for more information about memory allocation.

The following table shows the type of memory and the page designation each section type requires:

Section	Type of Memory	Page
.text	ROM or RAM	0
.cinit	ROM or RAM	0
.const	ROM or RAM	1
.switch	ROM or RAM	0
.bss	RAM	1
.stack	RAM	1
.sysmem	RAM	1

3.3.3 Sample Linker Command File

Example 3–1 shows a typical linker command file that can be used to link a C program. The command file in this example is named link.cmd.

Example 3–1. An Example of a Linker Command File

```
/************************************************/
/        Linker command file link.cmd            */
/************************************************/

-c               /* ROM autoinitialization model */
-m example.map   /* Create a map file            */
-o example.out   /* Output file name             */

main.obj         /* First C module               */
sub.obj          /* Second C module              */
asm.obj          /* Assembly language module     */
-l rts.lib       /* Runtime-support library      */
-l matrix.lib    /* Object library               */

MEMORY
{
   PAGE 0 : PROG: origin = 30h,    length = 0EFD0h
   PAGE 1 : DATA: origin = 800h    length = 0E800h
}
SECTIONS
{
    .text   > PROG  PAGE 0
    .cinit  > PROG  PAGE 0
    .switch > PROG  PAGE 0
    .bss    > DATA  PAGE 1
    .const  > DATA  PAGE 1
    .sysmem > DATA  PAGE 1
    .stack  > DATA  PAGE 1
}
```

First, the command file in Example 3–1 lists several linker options:

–c tells the linker to use the ROM model of autoinitialization.

–m tells the linker to create a map file; the map file in this example is named example.map.

–o tells the linker to create an executable object module; the module in this example is named example.out.

Next, the command file lists all the object files to be linked. This C program consists of two C modules, main.c and sub.c, which were compiled and assembled to create two object files called main.obj and sub.obj. This example also links in an assembly language module called asm.obj.

One of these files must define the symbol *main*, because boot.obj calls main as the start of your C program. All of these single object files are linked.

Finally, the command file lists all the object libraries that the linker must search. (The libraries are specified with the –l linker option.) Because this is a C program, the runtime-support library, rts.lib, must be included. Note that only the library members that resolve undefined references are linked.

To link the program, enter:

```
lnk500 link.cmd
```

The MEMORY directive, and possibly the SECTIONS directive, might require modification to work with your system. Refer to the *Linker Description* chapter in the *TMS320C54x Assembly Language Tools User's Guide* for information on these directives.

Chapter 4

TMS320C54x C Language Implementation

The C language that the TMS320C54x supports is 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*, by Kernighan and Ritchie. The second edition of *The C Programming Language* is based on the ANSI standard and is 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 TMS320C54x C compiler follows the ANSI standard for C. 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 TMS320C54x C compiler.

Topic **Page**

4.1 Characteristics of TMS320C54x 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 'C54x C compiler.

The following list identifies all such cases and describes the behavior of the 'C54x C compiler in each case. Each description also includes a reference to the formal ANSI standard and to *The C Programming Language* (second edition) by Kernighan and Ritchie (K&R).

4.1.1 Identifiers and Constants

The following conventions apply for identifiers and constants:

- The first 100 characters of all identifiers are significant. Case is significant; uppercase and lowercase characters are distinct for identifiers. These characteristics apply to all identifiers, internal and external, in all TMS320C54x tools. (ANSI 3.1.2, K&R A2.3)
- The source (host) and execution (target) character sets are assumed to be ASCII. There are no multibyte characters. (ANSI 2.2.1, K&R A12.1)
- Hexadecimal or octal escape sequences in character or string constants may have values up to 32 bits. (ANSI 3.1.3.4, K&R A2.5.2)
- Character constants with multiple characters are encoded as the last character in the sequence. For example,

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

4.1.2 Data Types

The following conventions apply for data types:

- For information about the representation of data types, see Section 4.2, *Data Types*, on page 4-4. (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)

4.1.3 Conversions

The following conventions apply for conversions:

- Float-to-integer conversions truncate toward 0. (ANSI 3.2.1.3, K&R A6.3)
- Pointers and integers can be freely converted. (ANSI 3.3.4, K&R A6.6)

4.1.4 Expressions

The following conventions apply for expressions:

- When two signed integers are divided and either is negative, the quotient (/) is negative, and the sign of the remainder (%) is the same as the sign of the numerator. For example,

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

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

4.1.5 Declarations

The following conventions apply for declarations:

- The *register* storage class is effective for all character, short, integer, and pointer types. (ANSI 3.5.1, K&R A8.1)
- Structure members are not packed into words (with the exception of bit fields). Each member is aligned on a 16-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)

4.1.6 Preprocessor

The preprocessor recognizes certain pragma directives; all others are ignored. Pragma directives tell the compiler's preprocessor how to treat functions. The recognized pragmas are described in Section 4.6 on page 4-9. (ANSI 3.8.6, K&R A12.8)

4.2 Data Types

The following information applies to data types:

- All integral types (char, short, int, and their unsigned counterparts) are equivalent types and are represented as 16-bit binary values.
- Long and unsigned long types are represented as 32-bit binary values.
- Signed types are represented in 2s-complement notation.
- The type char is a signed type, equivalent to int.
- Objects of type enum are represented as 16-bit values; in expressions, the type enum is equivalent to int.
- All floating-point types (float, double, and long double) are equivalent and are represented as IEEE single-precision format.

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

Table 4–1. TMS320C54x C Data Types

			Range	
Type	**Size**	**Representation**	**Minimum**	**Maximum**
char, signed char	16 bits	ASCII	–32768	32767
unsigned char	16 bits	ASCII	0	65535
short	16 bits	2s complement	–32768	32767
unsigned short	16 bits	Binary	0	65535
int, signed int	16 bits	2s complement	–32768	32767
unsigned int	16 bits	Binary	0	65535
long, signed long	32 bits	2s complement	–2147483648	214783647
unsigned long	32 bits	Binary	0	4294967295
enum	16 bits	2s complement	–32768	32767
float	32 bits	’C54x	1.19209290e–38	3.4028235e+38
double	32 bits	’C54x	1.19209290e–38	3.4028235e+38
long double	32 bits	’C54x	1.19209290e–38	3.4028235e+38
pointers	16 bits	Binary	0	0xFFFF

Many of the range values are available as standard macros in the header file limits.h, which is supplied with the compiler. For more information, see Section 6.7, *float.h and limits.h—Limits*, on page 6-17.

Note: 'C54x Byte Is 16 Bits

By ANSI C definition, the sizeof operator yields the number of bytes required to store an object. ANSI further stipulates that when sizeof is applied to char, the result is 1. Since the 'C54x char is 16 bits (to make it separately addressable), a byte is also 16 bits. This yields results you may not expect; for example, sizeof (int) = = 1 (not 2). 'C54x bytes and words are equivalent (16 bits).

4.3 Register Variables

The C compiler uses up to two register variables within a function. You must declare the register variables in the argument list or in the first block of the function. Register declarations in nested blocks are treated as normal variables.

The compiler uses AR1 and AR6 for register variables:

- AR1 is assigned to the first register variable.
- AR6 is assigned to the second variable.

The address of the variable is placed into the allocated register to simplify access. Thus, 16-bit types (char, short, int, and pointers) may be used as register variables.

Setting up a register variable at run time requires approximately four instructions per register variable. To use this feature efficiently, use register variables only if the variable is accessed more than twice.

The optimizer also creates register variables, but it uses them in a different way.

4.4 The asm Statement

The ’C54x C compiler allows you to imbed ’C54x 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 a single string-constant argument:

asm(”*assembler text***”);**

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

```
asm(”STR: .string \”abc\””);
```

The inserted code must be a legal assembly language statement. Like all assembly language statements, the line *must* begin with a label, a blank, a tab, or a comment (asterisk or semicolon). The compiler performs no checking on the string; if there is an error, it is detected by the assembler. For more information on assembly statements, refer to the *TMS320C54x Assembly Language Tools User’s Guide*.

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

Note: Avoid Disrupting the C 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 compiler cannot remove asm statements (except where such statements are totally unreachable), it can significantly rearrange the code order near asm statements, possibly causing undesired results. The asm command is provided so that you can access hardware features, which, by definition, C is unable to access.

4.5 Creating Global Register Variables

The ’C54x compiler extends the C language by adding a special convention to the register keyword to allow the allocation of global registers. In this special case, the register keyword is treated as a storage class modifier. The declaration must appear before any function definitions. This special declaration has the form:

register *type* **AR1**
or
register *type* **AR6**

4.5.1 About the Registers

The two registers AR1 and AR6 are normally save-on-entry registers; *type* cannot be float or long. When you use the allocation declaration at file level, the register is permanently reserved from any other use by the optimizer and code generator for that file. You cannot assign an initial value to the register. You can use a #define C statement to assign a meaningful variable name to the register and use the variable normally; for example:

```
register struct data_struct *AR6
#define data_pointer (AR6)

data_pointer–>element;
data_pointer++;
```

The following are reasons for using a global register variable:

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

You need to carefully consider the implications of assigning a global register variable. Since there are so few registers available, using this feature indiscriminately may result in inefficient code.

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

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

- Do not access a global register variable in an interrupt service routine unless you recompile all code, including all libraries, to reserve the register.
- Do not call functions that alter global register variables by functions that are not aware of the global register. You must be careful if you pass a pointer to a function as an argument. If the passed function alters the global register variable and the called function saves the register, the value in the register will be corrupted.
- Save the global register on entry into a module that uses it, and restore the register at exit.
- Determine whether the longjmp() function restores global register variables to the values they had at the setjmp() location. If this presents a problem in your code, you must unarchive the source for longjmp from rts.src and modify it.

4.5.2 Disabling the Compiler From Using AR1 and AR6

The –r*register* compiler option for the cl500 shell and the corresponding –g*register* option for the optimizer and code generator (if you are invoking the tools individually) prevent the compiler from using the named *register*. This lets you reserve the named register in modules (such as the runtime-support libraries) that do not have the global register variable declaration if you need to compile the modules to prevent some of the above occurrences.

You can disable the compiler's use of AR1 and AR6 completely so that you can use AR1 and/or AR6 in your interrupt functions without preserving them. If you disable the compiler from using AR1 and AR6, you must compile all code with the –r option(s) and rebuild the runtime-support library. For example, the following command rebuilds the rts.lib library to not use AR1 and AR6:

```
mk500 –rAR1 –rAR6 –o rts.src –l rts.lib
```

4.6 Pragma Directives

Pragma directives tell the compiler's preprocessor how to treat functions. The 'C54x C compiler supports the following pragmas:

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

The arguments *func* and *symbol* cannot be defined or declared inside the body of a function. You must specify the pragma outside the body of a function; and it must occur before any declaration, definition, or reference to the *func* or *symbol* argument. If you do not do this, the compiler issues a warning.

4.6.1 The CODE_SECTION Pragma

The CODE_SECTION pragma allocates space for the *symbol* in a section named *section name*. The *section name* is limited to a maximum of eight characters. The syntax of the pragma is:

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

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

Example 4–1 demonstrates the use of the CODE_SECTION pragma.

Example 4–1. Using the CODE_SECTION Pragma

(a) C source file

```
char bufferA[80];
char bufferB[80];

#pragma CODE_SECTION(funcA, ”codeA”)

char funcA(int i);
char funcB(int i);

void main()
{
   char c;
      c = funcA(1);
      c = funcB(2);
}

char funcA (int i)
{
   return bufferA[i];
}

char funcB (int j)
{
   return bufferB[j];
}
```

(b) Assembly source file

```
;******************************************************
;* TMS320C54x ANSI C Codegen         Version x.xx     *
;* Date/Time created: Mon Sep  9 12:07:42 1996        *
;******************************************************
       .mmregs
FP     .set      AR7
       .c_mode
;      ac500 pragma.c pragma.if
       .sect     ”.text”
       .global _main

;******************************************************
;* FUNCTION DEF: _main                                *
;******************************************************
_main:
       FRAME     #–1
       LD        #1,A
       CALL      #_funcA
       ; call occurs [#_funcA]
       STL       A,*SP(0)
       LD        #2,A
```

(b) Assembly source file (continued)

```
        CALL      #_funcB
        ; call occurs [#_funcB]
        STL       A,*SP(0)
        FRAME     #1
        RET
        ; return occurs


        .sect     ”codeA”
        .global   _funcA

;*****************************************************
;* FUNCTION DEF: _funcA                              *
;*****************************************************
_funcA:
        PSHM      AR1
        FRAME     #–1
        nop
        STL       A,*SP(0)
        MVDK      *SP(0),*(AR1)
        FRAME     #1
        LD        *AR1(_bufferA),A
        POPM      AR1
        RET
        ; return occurs

        .sect     ”.text”
        .global   _funcB

;*****************************************************
;* FUNCTION DEF: _funcB                              *
;*****************************************************
_funcB:
        PSHM      AR1
        FRAME     #–1
        nop
        STL       A,*SP(0)
        MVDK      *SP(0),*(AR1)
        FRAME     #1
        LD        *AR1(_bufferB),A
        POPM      AR1
        RET
        ; return occurs

        .global   _bufferA
        .bss      bufferA,80,0,0
        .global   _bufferB
        .bss      _bufferB,80,0,0
```

4.6.2 The DATA_SECTION Pragma

The DATA_SECTION pragma allocates space for the *symbol* in a section named *section name*. The *section name* is limited to a maximum of eight characters. The syntax of the pragma is:

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

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

Example 4–2 demonstrates the use of the DATA_SECTION pragma.

Example 4–2. Using the DATA_SECTION Pragma

(a) C source file

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

(b) Assembly source file

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

4.6.3 The FUNC_CANNOT_INLINE Pragma

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

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

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

The argument *func* is the name of the C function that cannot be inlined. For more information, see Section 2.4, *Function Inlining*, on page 2-40.

4.6.4 The FUNC_EXT_CALLED Pragma

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

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

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

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

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

4.6.5 The FUNC_IS_PURE Pragma

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

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

The pragma must appear before any declaration or reference to the function. The syntax of the pragma is:

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

The argument *func* is the name of a C function.

4.6.6 The FUNC_IS_SYSTEM Pragma

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

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

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

The argument *func* is the name of the C function to treat as an ANSI standard function.

4.6.7 The FUNC_NEVER_RETURNS Pragma

The FUNC_NEVER_RETURNS pragma specifies to the optimizer that the function never returns to its caller.

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

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

The argument *func* is the name of the C function that does not return.

4.6.8 The FUNC_NO_GLOBAL_ASG Pragma

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

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

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

The argument *func* is the name of the C function that makes no assignments.

4.6.9 The FUNC_NO_IND_ASG Pragma

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

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

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

The argument *func* is the name of the C function that makes no assignments.

4.6.10 The INTERRUPT Pragma

The INTERRUPT pragma enables you to handle interrupts directly with C code. The argument *func* is the name of a function. The pragma syntax is:

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

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

4.7 Initializing Static and Global Variables

The ANSI C standard specifies that both static and global (extern) variables without explicit initializations must be preinitialized to 0 before the program begins running. This task is typically performed when the program is loaded. Because the loading process depends heavily on the specific environment of the target application system, the compiler itself does not preinitialize variables; therefore, it is up to your application to fulfill this requirement.

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

```
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.

4.7.1 Initializing Static and Global Variables With the Const Type Qualifier

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

const static and global variables without explicit initializations are similar to other static and global variables because they may not be preinitialized to 0 (for the same reasons discussed in Section 4.7, *Initializing Static and Global Variables*). For example:

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

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

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

corresponds to an entry in the .const section:

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

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

4.7.2 Accessing I/O Port Space

The ioport keyword enables access to the I/O port space of the ’C54x devices. The keyword has the form:

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

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

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

For example, the following code declares the I/O port as unsigned port 10h, writes a to port 10h, then reads port 10h into b:

```
ioport unsigned port10; /* variable to access I/O port 10h */

int func ()
{
    ...

    port10 = a;         /* write a to port 10h            */
    ...

    b = port10;         /* read port 10h into b           */
    ...
}
```

The use of port variables is not limited to assignments. Port variables can be used in expressions like any other variable. Following are examples:

```
call(port10);    /* read port 10h and pass to call          */
a = port10 + b;  /* read port 10h, add b, assign to a       */
port10 += a;     /* read port 10h, add a, write to port 10h */
```

4.8 Compatibility with K&R C

The ANSI C language is a superset of the de facto C standard defined in K&R. Most programs written for other non-ANSI compilers correctly compile and run without modification.

However, there are subtle changes in the language that may affect existing code. Appendix C in K&R summarizes the differences between ANSI C and the first edition of K&R.

To simplify the process of compiling existing C programs with the 'C54x ANSI C compiler, the compiler has a K&R option (–pk) that modifies some semantic rules of the language for compatibility with older code. In general, the –pk option relaxes requirements that are stricter for ANSI C than for K&R C. The –pk option does not disable any new features of the language such as function prototypes, enumerations, initializations, or preprocessor constructs. Instead, –pk simply liberalizes the ANSI rules without revoking any of the features.

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

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

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

  Even without –pk, a violation of this rule is a code-E (recoverable) error. The –pe option, which converts code-E errors to warnings, can be used as an alternative to –pk.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.9 Compiler Limits

Due to the variety of host systems supported by the ’C54x 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 4–2 specifies the limits that are absolute. All the absolute limits equal or exceed those required by the ANSI C standard. When two values are listed, the first is for PC host systems, and the second is for other hosts. All the absolute limits equal or exceed those mandated 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.

Table 4–2. Absolute Compiler Limits

Description	Limits
Filename length	512 characters
Source line length; reflects the number of characters after splicing of \ lines; applies to any single macro definition or invocation	16K characters
Length of strings built from # or ##; reflects the number of characters before concatenation; all other character strings are unrestricted	512 characters
Macro definitions	Allocated from available system memory
Macros predefined with –d	64
Macro parameters	32 parms
Macro nesting; includes argument substitutions	32 levels
#include search paths; includes –i and C_DIR directories	64 paths
#include file nesting	64 levels
Conditional inclusion (#if) nesting	64 levels
Nesting of struct, union, or prototype declarations	20 levels
Function parameters	48 parms
Array, function, or pointer derivations on a type	12 derivations
Aggregate initialization nesting	32 levels
Static initializers	1500 per initialization (approximately)
Local initializers	150 levels (approximately)
Nesting of declarations in structures, unions, or protoypes	32 levels
Global symbols; may be further limited by available system memory	2000 PCs 10000 All others
Block scope symbols visible at any point	500 PCs 1000 All others
Number of unique string constants	400 PCs 1000 All others
Number of unique floating-point constants	400 PCs 1000 All others

Chapter 5

Runtime Environment

This section describes the TMS320C54x C runtime environment. To ensure successful execution of C programs, it is critical that all runtime code maintain this environment. Without the proper environment, your code is not reliable. It is also important to follow the guidelines in this chapter if you write assembly language functions that interface to C code.

Topic **Page**

5.1 Memory Model

The ’C54x treats memory as two linear blocks of program memory and data memory:

- **Program memory** contains executable code.
- **Data memory** contains external variables, static variables, and the system stack.

Blocks of code or data generated by a C program are placed into contiguous blocks in the appropriate memory space.

Note: The Linker Defines the Memory Map

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

5.1.1 Sections

The compiler produces seven 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 COFF sections, refer to Chapter 2 of the *TMS320C54x Assembly Language Tools User’s Guide*.

There are two basic types of sections:

- **Initialized sections** contain data tables or executable code. The C compiler creates four initialized sections: .text, .cinit, .const, and .switch.
  - The **.text section** is an initialized section that contains all the executable code as well as constants.
  - The **.cinit section** is an initialized section that contains tables for initializing variables and constants.
  - The **.const section** is an initialized section that contains string constants, and the declaration and initialization of global and static variables (qualified by *const*) that are explicitly initialized.
  - The **.switch section** is an initialized section that contains tables for switch statements.

- **Uninitialized sections** reserve space in memory (usually RAM). A program can use this space at runtime for creating and storing variables. The compiler creates three uninitialized sections: .bss, .stack, and .sysmem.
  - The **.bss section** is an uninitialized section that reserves space for global and static variables. At program startup time, the C boot routine copies data out of the .cinit section (which may be in ROM) and stores it in the .bss section.
  - The **.stack section** is an uninitialized section used for the C system stack. This memory is used to pass arguments to functions and to allocate space for local variables.
  - The **.sysmem section** is a uninitialized section that reserves space for dynamic memory allocation. The reserved space is used by malloc functions. If no malloc functions are used, the size of the section remains 0.

The *assembler* creates an additional section called .data, which it usually uses for initialized data. The C compiler does not use the .data section.

The linker takes the individual sections from different modules and combines sections with the same name to create eight output sections. The complete program is made up of these eight output sections, which include the assembler's .data section. You can place these output sections anywhere in the address space, as needed, to meet system requirements.

The .text, .cinit, and .switch sections are usually linked into either ROM or RAM, and must be in program memory (page 0). The .const section can also be linked into either ROM or RAM but must be in data memory (page 1). The .bss, .stack, and .sysmem sections must be linked into RAM and must be in data memory. The following table shows the type of memory and page designation each section type requires:

Section	Type of Memory	Page
.text	ROM or RAM	0
.cinit	ROM or RAM	0
.switch	ROM or RAM	0
.const	ROM or RAM	1
.bss	RAM	1
.stack	RAM	1
.sysmem	RAM	1

For more information about allocating sections into memory, see the *TMS320C54x Assembly Language Tools User's Guide*.

5.1.2 C System Stack

The C compiler uses a stack to:

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

The runtime stack grows down from high addresses to lower addresses. The compiler uses the hardware stack pointer (SP) to manage the stack.

The stack size is set by the linker. The linker also creates a global symbol, __STACK_SIZE, and assigns it a value equal to the size of the stack in bytes. The default stack size is 1K words. You can change the size of the stack at link time by using the –stack option on the linker command line and specifying the size of the stack as a constant immediately after the option.

5.1.3 Allocating .const to Program Memory

If your system configuration does not support allocating an initialized section such as .const to data memory, then you have to allocate the .const section to load in program memory and run in data memory. Then at boot time, copy the .const section from program to data memory. The following sequence shows how you can perform this task:

Modify the boot routine:

1) Extract boot.asm from the source library:

```
ar500 –x rts.src boot.asm
```

2) Edit boot.asm and change the CONST_COPY flag to 1:

```
CONST_COPY .set  1
```

3) Assemble boot.asm:

```
asm500 boot.asm
```

4) Archive the boot routine into the object library:

```
ar500 –r rts.lib boot.obj
```

Link with a linker command file that contains the following entries:

```
MEMORY
{
   PAGE 0 : PROG : ...
   PAGE 1 : DATA : ...
}

SECTIONS
{
   ...
   .const  : load = PROG PAGE 1, run = DATA PAGE 1
             {
                /* GET RUN ADDRESS   */
                 __const_run = .;
                /* MARK LOAD ADDRESS */
                 *(.c_mark)
                /* ALLOCATE .const   */
                 *(.const)
                /* COMPUTE LENGTH    */
                 __const_length = .– __const_run;
              }
   ...
}
```

In your linker command file, you can substitute the name PROG with the name of a memory area on page 0 and DATA with the name of a memory area on page 1. The rest of the command file must use the names as above. The code in boot.asm that is enabled when you change CONST_COPY to 1 depends on the linker command file using these names in this manner. To change any of the names, you must edit boot.asm and change the names in the same way.

5.1.4 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. This is accomplished by declaring a large memory pool, or heap, and then using the functions to allocate memory from the heap. Dynamic allocation is not a standard part of the C language; it is provided by standard runtime-support functions.

This memory pool, or heap, is created by the linker. The linker also creates a global symbol, __SYSMEM_SIZE, and assigns it a value equal to the size of the heap in words. The default heap size is 1K words. You can change the size of the memory pool at link time with the –heap option. Specify the size of the memory pool as a constant after the –heap option on the linker command line.

5.1.5 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 (in ROM) to the initialized variables in .bss (RAM).

In situations where a program is loaded directly from an object file into memory and then run, you can avoid having the .cinit section occupy space in memory. A loader can read the initialization tables directly from the object file (instead of from ROM) and perform the initialization directly at load time (instead of at runtime). You can specify this *to the linker* by using the –cr linker option. For more information on the –cr linker option, see subsection 3.1.2, *The –c and –cr Linker Options*, on page 3-3. For more information on system initialization, see Section 5.8, *System Initialization*, on page 5-26.

5.1.6 Allocating Memory for Static and Global Variables

A unique, contiguous space is allocated for all external or static variables declared in a C program. The linker determines the address of the space. The compiler ensures that space for these variables is allocated in multiples of words so that each variable is aligned on a word boundary.

The C compiler expects global variables to be allocated into data memory. (It reserves space for them in .bss.) Variables declared in the same module are allocated into a single, contiguous block of memory.

5.1.7 Field/Structure Alignment

When the compiler allocates space for a structure, it allocates as many words as are needed to hold all of the structure's members. In an array of structures, each structure begins on a word boundary.

All nonfield types are aligned on word boundaries. Fields are allocated as many bits as requested. Adjacent fields are packed into adjacent bits of a word, but they do not overlap words; if a field would overlap into the next word, the entire field is placed into the next word. Fields are packed as they are encountered; the most significant bits of the structure word are filled first.

5.1.8 Character String Constants

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

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

  ```
  char s[] = ”abc”;
  ```

  When a string is used as an initializer, it is simply treated as an initialized array; each character is a separate initializer. For more information, see Section 5.8, *System Initialization*, on page 5-26.

- 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 included. For example, the following lines define the string abc, along with the terminating byte; the label SL5 points to the string:

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

  String labels have the form SL*n*, where *n* is a number assigned by the compiler to make the label unique. These numbers begin at 0 with an increase of 1 for each defined string. All strings used in a source module are defined at the end of the compiled assembly language module.

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

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

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

```
char *a = ”abc”;
a[1] = ’x’;        /* Incorrect! */
```

5.2 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, you must follow these register conventions.

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

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.

Table 5–1 summarizes how the compiler uses the 'C54x registers and shows which registers are defined to be preserved across function calls.

Table 5–1. Register Use and Preservation Conventions

Register(s)	Usage	Save on Entry	Save on Call
AR0	Pointers and expressions	No	Yes
AR1	Pointers and expressions	Yes	No
AR2 – AR5	Pointers and expressions	No	Yes
AR6	Pointers and expressions	Yes	No
AR7	Pointers, expressions, frame pointer (when needed)	Yes	No
A	Expressions, passes first argument to functions, returns result from functions	No	Yes
B	Expressions	No	Yes
SP	Stack pointer	†	†
T	Multiply and shift expressions	No	Yes
ST0, ST1	Status registers	No	Yes
BRC	Block repeat counter	No	Yes

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

5.2.1 Status Registers

Table 5–2 shows all of the status fields used by the compiler. Presumed value is the value the compiler expects in that field upon entry to, or return from, a function; a dash in this column indicates the compiler does not expect a particular value. The modified column indicates whether code generated by the compiler ever modifies this field.

Table 5–2. Status Register Fields

Field	Name	Presumed Value	Modified
ARP	Auxiliary register pointer	0	Yes
ASM	Accumulator shift mode	–	Yes
BRAF	Block repeat active bit	–	No
C	Carry bit	–	Yes
C16	Dual 16-bit math bit	0	No
CMPT	Compatibility mode bit	0	No
CPL	Compiler mode bit	1	No
FRCT	Fractional mode bit	0	No
OVA	Overflow flag for A	–	Yes
OVB	Overflow flag for B	–	Yes
SXM	Sign extension mode	–	Yes
TC	Test control bit	–	Yes

All other fields are not used and do not affect code generated by the compiler.

5.2.2 Register Variables

Register variables are local variables or compiler temporaries defined to reside in a register rather than in memory. The way the compiler uses registers for register variables is different depending on whether you use the optimizer.

5.2.2.1 Register variables when the optimizer is not used

When the optimizer is not used, the compiler allocates registers for up to two variables declared with the register keyword. You must declare the variables in the argument list or in the first block of the function. Register declarations in nested blocks are treated as normal variables.

The compiler uses AR1 and AR6 for these register variables. AR1 is allocated to the first variable, and AR6 is allocated to the second.

The *address* of the variable is placed into the allocated register to simplify access. Only 16-bit types (char, short, int, and pointers) may be used as register variables.

Setting up a register variable at runtime requires approximately four instructions per register variable. To use this feature efficiently, use register variables only if the variable will be accessed more than twice.

5.2.2.2 Register variables when the optimizer is used

When the optimizer is used, all user register declarations are ignored. The optimizer makes all the decisions on what variables or compiler temporaries are allocated to registers. The optimizer allocates the variables, not their addresses, directly to registers. The optimizer may allocate the registers AR1 and AR6 for register variables. Because AR5 is not preserved across function calls, it is not used for variables that overlap any calls.

Because the register use for variables depends on whether you use the optimizer, you should avoid writing code that depends on specific registers allocated to specific variables.

Note: Using AR1 and AR6 as Global Register Variables

If you have disabled the compiler from using AR1 and AR6 with the –r option, AR1 and AR6 are not available for use as register variables. See subsection 4.5.2, *Disabling the Compiler From Using AR1 and AR6*, on page 4-8, for more information.

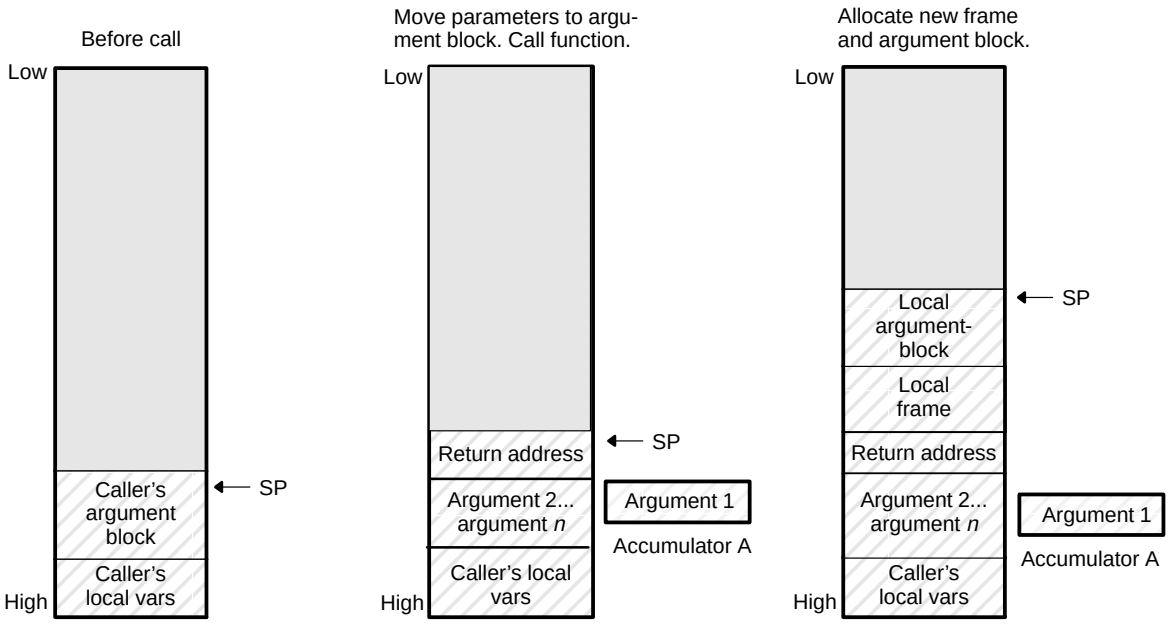
5.3 Function Calling Conventions

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 and calls another function. Note that the first parameter is passed in accumulator A. This example also shows allocation of a local frame and argument block for the called function. Functions that have no local variables and do not require an argument block do not allocate a local frame.

The term *argument block* refers to the part of the local frame used to pass arguments to other functions. Parameters are passed to a function by moving them into the argument block rather then pushing them on the stack. The local frame and argument block are allocated at the same time.

Figure 5–1. Use of the Stack During a Function Call



5.3.1 Calling a Function

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

1) The caller places the first (left-most) argument in accumulator A. The caller moves the remaining arguments to the argument block in reverse order, the leftmost remaining argument at the lowest address. Thus this argument is at the top of the stack when the function is called.

   Declaring a function with an ellipsis indicates that it can be called with a variable number of arguments. When a function is declared with an ellipsis and only one argument is explicitly declared, the convention requires the caller to pass this argument on the stack, not in accumulator A. This is so that the argument's stack address can act as a reference for accessing the undeclared arguments. For example:

```
int vararg(int first, ...); /* 'first' is passed */
                            /*  on the stack     */
```

2) 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 accumulator A.

3) The caller calls the function.

5.3.2 Responsibilities of a Called Function

A called function must perform the following tasks:

1) If the called function modifies AR1, AR6, or AR7, it pushes them on the stack.

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

   size of local variables + *max*

   The max value is the size of the parameters placed in the argument block for each call.

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

4) If the function returns a value, the called function places the value in accumulator A.

   If the function returns a structure, the called function copies the structure to the memory block that accumulator A points to. If the caller does not use the return value, A 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 A and call f. Function f then copies the return structure directly into s, performing the assignment automatically.

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

5) The called function deallocates the frame and argument block by adding the constant computed in step 2.

6) The called function restores all saved registers.

7) The called function executes a return.

For example:

```
callee:                 ; entry point to the function
   PSHM   AR6           ; save AR6
   PSHM   AR7           ; save AR7
   FRAME  #–15          ; allocate frame and
                        ; argument block

   ...                  ; body of the function

   FRAME  #15           ; deallocate the frame and
                        ; argument block
   POPM   AR7           ; restore AR7
   POPM   AR6           ; restore AR6
   RET                  ; return
```

5.3.3 Accessing Arguments and Locals

The compiler uses the compiler mode (selected when the CPL bit in status register ST1 is set to 1) for accessing arguments and locals. When this bit is set, the direct addressing mode computes the data address by adding the constant in the dma field of the instruction to the SP. For example:

```
ADD       *SP(4), A    ; A += *(SP+4)
```

The largest offset available with this addressing mode is 127. So, if an object is too far away from the SP to use this mode of access, the compiler copies the SP to AR0 in the function prolog, then uses long offset addressing to access the data. For example:

```
MVMM      SP, AR0      ; AR0 = SP (in prolog)
...
   ADD       *AR0(129),A  ; A += *(AR0 + 129)
```

5.3.4 Allocating the Frame and Using the 32-bit Memory Read Instructions

Some ’C54x instructions read and write 32 bits of memory at once (DLD, DADD, etc.). For more information on how these instructions access memory, see the *TMS320C54x User’s Guide*. As a result, the compiler must ensure that all 32-bit objects reside at even word boundaries. To ensure that this occurs, the compiler takes these steps:

1) It initializes the SP to an even boundary.

2) Because a CALL instruction subtracts 1 from the SP, it assumes that the SP is odd at function entry.

3) It makes sure that the number of PSHM instructions plus the number of words allocated with the FRAME instruction totals an odd number, so that the SP points to an even address.

4) It makes sure that 32-bit objects are allocated to even addresses, relative to the known even address in the SP.

5) Because interrupts cannot assume that the SP is odd or even, it aligns the SP to an even address.

5.4 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 5.4.1, *Assembly Language Modules*). This is the most versatile method.
- Use inline assembly language, embedded directly in the C source (see subsection 5.4.3, *Inline Assembly Language*, on page 5-19).
- Modify the assembly language code that the compiler produces (see subsection 5.4.4, *Modifying Compiler Output*, on page 5-20).

5.4.1 Assembly Language Modules

Interfacing with assembly language functions is straightforward if you follow the calling conventions defined in Section 5.3, *Function Calling Conventions*, on page 5-11, and the register conventions defined in Section 5.2, *Register Conventions*, on page 5-8. 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 5.2, *Register Conventions*, on page 5-8.
- Dedicated registers modified by a function must be preserved. Dedicated registers include:

  AR1
  AR6
  AR7
  SP

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

  Any register that is not dedicated can be used freely without being preserved.

- Interrupt routines must save *all* the registers they use. (For more information about interrupt handling, see Section 5.5, *Interrupt Handling*, on page 5-21.)

- When calling a C function from assembly language, the first (leftmost) argument must be placed in accumulator A. The remaining arguments should be placed on the stack in reverse order. That is, the rightmost argument at the highest (deeper in the stack) address. You can do this by either directly moving the arguments to an argument block on the stack like the compiler does, or you can push them.

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

  If the function you are calling accepts one defined argument and an undefined number of additional arguments, all the arguments must go on the stack. The first one does not go in accumulator A. See step 1 in subsection 5.3.1, *Calling a Function*, on page 5-12.

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

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

- Functions must return values in accumulator A. Structures are returned as described in step 4 in subsection 5.3.2, *Responsibilities of a Called Function*, on page 5-12.

- No assembly language 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 will cause unpredictable results.

- The compiler appends an underscore (_) to the beginning of all identifiers. In assembly language modules, you must use the prefix _ for all objects that are to be accessible from C. 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 an underscore may be safely used without conflicting with a C identifier.

- Any object or function declared in assembly language 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.

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

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

```
LD *(global_var), A   ; works with CPL == 1
LD global_var, A      ; doesn't work with CPL == 1
```

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

Example 5–1 illustrates a C function called main, which calls an assembly language function called asmfunc. The asmfunc function takes its single argument, adds it to the C global variable called gvar, and returns the result.

Example 5–1. An Assembly Language Function

(a) C program

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

main()
{
   int i;

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

(b) Assembly language program

```
_asmfunc:

   ADD  *(_gvar),A    ; add gvar to A => i is in A
   STL  A, *(_gvar)   ; return result in A
   RETD               ; start return
```

In the C program in Example 5–1, the extern declaration of asmfunc is optional, since the function returns an int. Like C functions, assembly functions need be declared only if they return noninteger values. In the assembly language code in Example 5–1, note the underscores on all the C symbol names used in the assembly code.

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

5.4.2 How to Define Variables in Assembly Language

It is sometimes useful for a C program to access variables defined in assembly language. Accessing uninitialized variables from the .bss section is straightforward:

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

Example 5–2 shows an example that accesses a variable defined in .bss.

Example 5–2. Accessing a Variable Defined in .bss From C

(a) C Program

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

(b) Assembly Language Program

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

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

You may not always want a variable to be in the .bss section. For example, a common situation is a lookup table defined in assembly language that you 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; it is helpful (but not necessary) to put it in its own initialized section. Declare a global label that points to the beginning of the object, and then the object can be linked anywhere into the memory space. To access it in C, you must declare the object as *extern* and not precede it with an underscore. Then you can access the object normally.

Example 5–3 shows an example that accesses a variable that is not defined in .bss.

Example 5–3. Accessing from C a Variable Not Defined in .bss

(a) C Program

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

(b) Assembly Language Program

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

5.4.3 Inline Assembly Language

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

Note: Using the asm Statement

The asm statement is provided so you can access features of the hardware that would be otherwise inaccessible from C. When you use the asm statement, be extremely careful not to disrupt the C environment. The compiler does not check or analyze the inserted instructions.

Inserting jumps or labels into C code may produce unpredictable results by confusing the register-tracking algorithms that the code generator uses.

Do not change the value of a C variable; however, you can safely read the current value of any variable.

Do not use the asm statement to insert assembler directives that change the assembly environment.

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”);
```

5.4.4 Modifying Compiler Output

You can inspect and change the assembly language output that the compiler produces by compiling the source and then editing the output file before assembling it. The C interlist utility is useful for inspecting compiler output (see Section 2.5, *Using the Interlist Utility*, on page 2-45).

The warnings in subsection 5.4.3, *Inline Assembly Language*, about disrupting the C environment also apply to modification of compiler output.

5.5 Interrupt Handling

As long as you follow the guidelines in this section, C code can be interrupted and returned to without disrupting the C environment. When the C environment is initialized, the startup routine does not enable or disable interrupts. (If the system is initialized via a hardware reset, interrupts are disabled.) If your system uses interrupts, it is your responsibility to handle any required enabling or masking of interrupts. Such operations have no effect on the C environment and can be easily implemented with asm statements.

5.5.1 General Points About Interrupts

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

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

- It is your responsibility to handle any special masking of interrupts (via the IMR register). You can use inline assembly to enable or disable the interrupts and modify the IMR register without corrupting the C environment or C pointer.
- An interrupt handling routine cannot have arguments. If any are declared, they are ignored.
- An interrupt handling routine can be called by normal C code, but it is inefficient to do this because all the registers are saved.
- An interrupt handling routine can handle a single interrupt or multiple interrupts. The compiler does not generate code that is specific to a certain interrupt, except for c_int00, which is the system reset interrupt. When you enter this routine, you cannot assume that the runtime stack is set up; therefore, you *cannot allocate local variables*, and you *cannot save any information on the runtime stack*.
- To associate an interrupt routine with an interrupt, a branch must be placed in the appropriate interrupt vector. You can use the assembler and linker to do this by creating a simple table of branch instructions using the .sect assembler directive.
- In assembly language, remember to precede the symbol name with an underscore. For example, refer to c_int00 as _c_int00.

5.5.2 Using C Interrupt Routines

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

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

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

5.5.3 Saving Context on Interrupt Entry

All registers that the interrupt routine uses, including the status registers, must be preserved. If the interrupt routine calls other functions, *all* of the registers in Table 5–1 on page 5-8 must be preserved.

Some ’C54x instructions access 32 bits of memory at once (DLD, DADD, etc.). As a result, the compiler must take steps to ensure that the stack pointer always contains an even value. These steps are detailed in subsection 5.3.4, *Allocating the Frame and Using the 32-bit Memory Read Instructions*, on page 5-14. (For more information on how these instructions access memory, refer to the *TMS320C54x User’s Guide*.)

Interrupt routines do not know whether the stack pointer is even or odd. Therefore, the compiler issues these instructions to save the registers and align the stack pointer.

```
PSHM  ST0           ; first save off all other registers
. . .
PSHM  SP            ; push the SP
ANDM  #0FFFEH,*(SP) ; align to even boundary
```

5.6 Integer Expression Analysis

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

5.6.1 Arithmetic Overflow and Underflow

The ’C54x produces a 40-bit result even when 16-bit or 32-bit values are used as data operands; thus, *arithmetic overflow and underflow cannot be handled in a predictable manner*. If your code depends on a particular type of overflow/underflow handling, there is no guarantee that this code will execute correctly.

5.6.2 Operations Evaluated With RTS Calls

The ’C54x does not directly support some C operations. Evaluating these operations is done with calls to runtime-support routines. These routines are hard-coded in assembly language. They are members of the object and source RTS libraries (rts.lib and rts.src) in the toolset.

The conventions for calling these routines are modeled on the standard C calling conventions. For binary routines (divide, etc.), the left operand is passed in accumulator A and the right operand is passed on the stack. The result is returned in accumulator A. For unary routines, the argument is passed and the result returned in accumulator A.

Operation Type	Operations Evaluated with RTS Calls
16-bit int	Divide
	Modulus
32-bit long	Divide
	Modulus
	Multiply
	Shift left
	Shift right

5.6.3 C Code Access to the Upper 16 Bits of 16-Bit Multiply

The following methods provide access to the upper 16 bits of a 16-bit multiply in C language:

- Signed-results method:

```
int m1, m2;
int result;
result = ((long) m1 * (long) m2) >> 16;
```

- Unsigned-results method:

```
unsigned m1, m2;
unsigned result;
result = ((unsigned long) m1 * (unsigned long) m2) >> 16;
```

Both result statements are implemented by the compiler without making a function call to the 32-bit multiply routine.

Note: Danger of Complicated Expressions

The compiler must recognize the structure of the expression in order for it to return the expected results. Avoid complicated expressions such as the following:

```
((long)((unsigned)((a*b)+c)<5)*(long)(z*sin(w)>6))>>16
```

5.7 Floating-Point Expression Analysis

The ’C54x C compiler represents floating-point values as IEEE single-precision numbers. Both single-precision and double-precision floating-point numbers are represented as 32-bit values; there is no difference between the two formats.

The ’C54x runtime-support library, rts.lib, contains a custom-coded set of floating-point math functions that support:

- Addition, subtraction, multiplication, and division
- Comparisons (>, <, >=, <=, ==, !=)
- Conversions from integer or long to floating-point and floating-point to integer or long, both signed and unsigned
- Standard error handling

The conventions for calling these routines are the same as the conventions used to call the integer operation routines. Conversions are unary operations.

5.8 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 runtime-support source library contains the source to this routine in a module called boot.asm.

The c_int00 function can be called by reset hardware to begin running the system. The function is in the runtime-support library (rts.lib) and must be linked with the C object modules. This occurs by default when you use the –c or –cr option (see subsection 3.1.1, *About the –c and –cr Linker Options*, on page 3-3) in the linker and include rts.lib (see Section 6.1, *About the Runtime–Support Libraries*, on page 6-2) as a linker input file. 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:

- Reserves space in .bss for the runtime stack and sets up the initial stack pointer
- Autoinitializes global variables by copying the data from the initialization tables in .cinit to the storage allocated for the variables in .bss. In the RAM autoinitialization model, a loader performs this step before the program runs (it is not performed by the boot routine).
- 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 three operations listed above in order to correctly initialize the C environment.

5.8.1 Runtime Stack

The runtime stack is allocated in a single contiguous block of memory and grows down from high addresses to lower addresses. The SP points to the top of the stack.

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

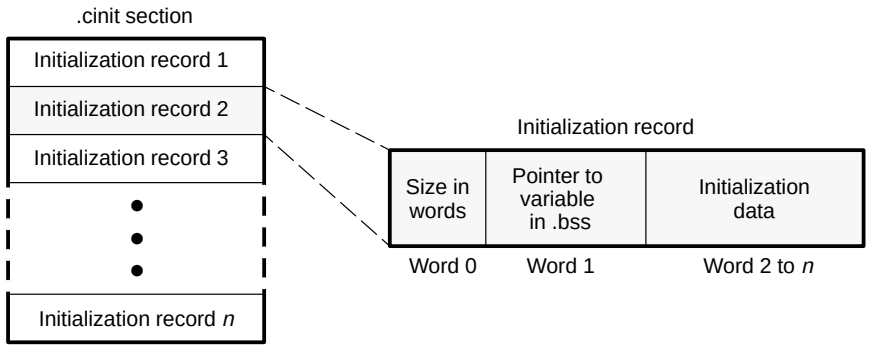
The stack size can be changed at link time by using the –stack option on the linker command line and specifying the stack size as a constant directly after the option.

5.8.2 Autoinitialization of Variables and Constants

Before program execution, any global variables declared as preinitialized must be initialized by the boot function. The compiler builds tables that contain data for initializing global and static variables in a .cinit section in each file. All compiled modules contain these initialization tables. The linker combines them into a single table, which is then used to initialize all the system variables. (Do not place any other data in the .cinit section; this corrupts the tables.)

The tables in the .cinit section consist of variable-size initialization records. Figure 5–2 shows the format of the .cinit section and the initialization records.

Figure 5–2. Format of Initialization Records in the .cinit Section



The fields of an initialization record contain the following information:

1) The first field (word 0) is the size in words of the initialization data for the variable.

2) The second field (word 1) 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.)

3) The third field (words 2 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 autoinitialized. 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 interface assembly language modules to your C programs, do not use the .cinit section for any other purpose.

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

Note that const-qualified variables are initialized differently; see Section 4.7, *Initializing Static and Global Variables*, on page 4-16.

5.8.2.1 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 can save the memory used by the initialization tables. The RAM option requires the use of a 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. 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.

5.8.2.2 Initializing Variables in the ROM Model

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

The linker defines a special symbol called cinit that points to the beginning of the 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.

Chapter 6

Runtime-Support Functions

Some of the tasks that a C program performs (such as memory allocation, string conversion, and string searches) are not part of the C language. 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. All of the ANSI functions except those that require an underlying operating system (such as signals) are provided.

If you use any of the runtime-support functions, be sure to build the appropriate library, according to the device, using the library-build utility (see Chapter 7, *Library-Build Utility*); include that library when you link your C program.

Topic **Page**

6.1 About the Runtime-Support Libraries

The following libraries are included with the ’C54x C compiler:

- *rts.lib* contains the ANSI runtime-support object library
- *rts.src* contains the source for the ANSI runtime-support routines

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

6.1.1 Linking Code with the Object Library

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 specify libraries *last* on the linker command line because the assembler searches for unresolved references when it encounters a library on the command line. When a library is linked, the linker includes only those library members required to resolve undefined references. For more information about linking, see Chapter 3, *Linking C Code*.

There is one header file, values.h, in rts.src, that is not a standard header. It is provided so that you can customize the functions. It contains definitions necessary for recompiling the trigonometric and transcendental math functions.

6.1.2 Modifying a Library Function

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

```
ar500 –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:

```
ar500 –r rts.lib atoi.obj strcpy.obj
```

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 7 of the *TMS320C54x Assembly Language Tools User’s Guide*.

6.1.3 Building a Library With Different Options

You can create a new library from rts.src by using the runtime-support installation utility, mk500. For example, use this command to build a fast, optimized runtime-support library:

```
mk500 –-u –o2 –x –ms rts.src –l rtsf.lib
```

The –-u option tells the mk500 utility to use the header files in the current directory, rather than extracting them from the source archive. The –o2 and –x options are the optimizer options used to build the rts.lib supplied with the toolset. The –ms option tells the compiler to optimize the code for space considerations.

6.2 About the 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:

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.

6.3 assert.h—Diagnostic Messages

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 filename, 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, assert is turned off and does nothing. If NDEBUG *is not* defined, assert is enabled.

The assert macro is defined as follows:

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

For a complete description, see the assert function on page 6-37.

6.4 ctype.h—Character Typing and Conversion

The *ctype.h* header contains declarations for the following:

- Functions that test a character to determine whether it is a letter, a printing character, a hexadecimal digit, and so on. These functions return a value of *true* (a nonzero value) or *false* (zero).
- Functions that convert characters to lowercase, uppercase, or ASCII and return the converted character.
- Macros that perform the same operations as the functions.

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 macros run faster than the corresponding functions. The macros expand to a lookup operation in an array of flags (the 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). If an argument passed to one of these macros has side effects, use the function instead. For example, _isdigit (*p++) has the effect of incrementing p. The macro may increment p more than once, but the function isdigit assures the correct result.

The following table lists the character typing functions and indicates the page that contains more detailed information about the function:

Function	Description	Page
isalnum	Tests character for an alphanumeric ASCII character	6-54
isalpha	Tests character for an alphabetic ASCII character	6-54
isascii	Tests character for an ASCII character	6-54
iscntrl	Tests character for a control character	6-54
isdigit	Tests character for a numeric character	6-54
isgraph	Tests character for any printing character except a space	6-54
islower	Tests character for a lowercase alphabetic ASCII character	6-54
isprint	Tests character for a printable ASCII character	6-54
ispunct	Tests character for an ASCII punctuation character	6-54
isspace	Tests character for an ASCII spacebar, tab, carriage return, formfeed, or newline character	6-54
isupper	Tests character for an uppercase ASCII alphabetic character	6-54
isxdigit	Tests character for a hexadecimal digit	6-54
toascii	Converts character to a legal ASCII value	6-82
tolower	Converts character to lowercase if it's uppercase	6-83
toupper	Converts character to uppercase if it's lowercase	6-83

6.5 errno.h—Error Reporting

Errors can occur in a math function if invalid parameter values are passed to the function or if the function returns a result that is outside the defined range for the type of the result. When this happens, a variable named errno is set to the value of one of the following macros:

- EDOM, for domain errors (invalid parameter)
- ERANGE, for range errors (invalid result)

C code that calls a math function can read the value of errno to check for error conditions. The errno variable is declared in *errno.h* and defined in errno.c.

6.6 file.h—Low-Level I/O Functions

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

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

6.6.1 Using the I/O Functions

To use the I/O functions:

- ❏ Include the header file stdio.h for each module that references a function.
- ❏ Invoke the debugger with the –o option to enable I/O support. For more information about the debugger –o option, see the *TMS320C54x C Source Debugger User's Guide*.

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

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

6.6.2 Example

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

```
#include <stdio.h>

main()
{
 FILE *fid;

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

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

Issue the following shell command to compile, link, and create the file main.out:

```
cl500 main.c –z –heap 400 –l rts.lib –o main.out
```

Execute main.out under the debugger on a SPARC host to accomplish the following:

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

6.6.3 Overview Of Low-Level I/O Implementation

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

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

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

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

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

At the next level are the user-definable device-level drivers. They map directly to the low-level I/O functions. The runtime-support library includes the device drivers necessary to perform I/O on the host on which the debugger is running.

The specifications for writing device-level routines to interface with the low-level routines are shown on pages 6-13 to 6-16. Each function must set up and maintain its own data structures as needed. Some function definitions perform no action and just return.

6.6.4 Adding A Device For C I/O

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

1) Define the device-level functions as described in subsection 6.6.3, *Overview Of Low-Level I/O Implementation*.

Note: Use Unique Function Names

The function names OPEN, CLOSE, READ, and so on, are used by the low-level routines. Use other names for the device-level functions that you write.

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

name	is the string for the device name.
flags	specify whether the device supports multiple streams or not.
function pointers	are pointers to the device-level functions: ❑ CLOSE ❑ LSEEK ❑ OPEN ❑ READ ❑ RENAME ❑ WRITE ❑ UNLINK

The first entry in the device table is predefined to be the host device on which the debugger is running. The low-level routine, add_device(), finds the first empty position in the device table and initializes the device fields with the passed-in arguments. For a complete description, see the add_device function on page 6-34.

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

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

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

main()
{
   FILE *fid;

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

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

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

   fclose(fid);
}
```

CLOSE — *Close File or Device For I/O*

Syntax **int CLOSE(int** *file_descriptor***);**

Description Closes the device or file associated with *file_descriptor*

Parameters

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

Return Value

Successful:	0
Unsuccessful:	–1

LSEEK — *Set File Position Indicator*

Syntax **long LSEEK(int** *file_descriptor***, long** *offset***, int** *origin***);**

Description Sets the file position indicator for the given file to *origin* + *offset*. The file position indicator measures the position in characters from the beginning of the file.

Parameters

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

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

Return Value

Successful:	The new value of the file-position indicator
Unsuccessful:	EOF

OPEN — *Open File or Device For I/O*

Syntax

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

Description

Opens the device or file specified by *path* and prepares it for I/O

Parameters

path — The filename of the file to be opened, including path information

flags — Attributes that specify how the device or file is manipulated. Specified using the following symbols:

```
O_RDONLY  (0x0000)  /* open for reading */
O_WRONLY  (0x0001)  /* open for writing */
O_RDWR    (0x0002)  /* open for read & write */
O_APPEND  (0x0008)  /* append on each write */
O_CREAT   (0x0100)  /* open with file create */
O_TRUNC   (0x0200)  /* open with truncation */
O_BINARY  (0x8000)  /* open in binary mode */
```

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

mode — Required but ignored

Return Value

Successful: The stream number assigned by the low-level routines that the device-level driver must associate with the opened file or device

Unsuccessful: < 0

READ — *Read Characters From Buffer*

Syntax **int READ(int** *file_descriptor***, char** **buffer***, unsigned** *count***);**

Description Reads the number of characters specified by *count* to the *buffer* from the device or file associated with *file_descriptor*.

Parameters

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

Return Value

Successful:	0 if EOF was encountered before the read was complete Number of characters read in every other instance
Unsuccessful:	–1

RENAME — *Rename File*

Syntax **int RENAME(char ***_old_name_**, char ***_new_name_**);**

Description Changes the name of a file

Parameters

old_name	The current name of the file
new_name	The new name for the file

Return Value

Successful:	0
Unsuccessful:	<> 0

UNLINK — *Delete File*

Syntax **int UNLINK(char ***_path_**);**

Description Deletes the file specified by *path*

Parameters

path	The path name of the file to delete

Return Value

Successful:	0
Unsuccessful:	–1

WRITE — *Write Characters to Buffer*

Syntax

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

Description

Writes the number of characters specified by *count* from the *buffer* to the device or file associated with *file_descriptor*

Parameters

file_descriptor	The stream number assigned by the low-level routines that is associated with the opened file or device
buffer	The location of the buffer the write characters are stored in
count	The number of characters to write to the device or file

Return Value

Successful: Number of characters written
Unsuccessful: –1

6.7 float.h and limits.h—Limits

The float.h and limits.h headers define macros that expand the useful limits and parameters of the ’C54x numeric representations. Table 6–1 and Table 6–2 list these macros and their associated limits.

Table 6–1. Macros That Supply Integer Type Range Limits (*limits.h*)

Macro	Value	Description
CHAR_BIT	16	Maximum number of bits for the smallest object that is not a bit field
SCHAR_MIN	–32 768	Minimum value for a signed char
SCHAR_MAX	32 767	Maximum value for a signed char
UCHAR_MAX	65 535	Maximum value for an unsigned char
CHAR_MIN	SCHAR_MIN	Minimum value for a char
CHAR_MAX	SCHAR_MAX	Maximum value for a char
SHRT_MIN	–32 768	Minimum value for a short int
SHRT_MAX	32 767	Maximum value for a short int
USHRT_MAX	65 535	Maximum value for an unsigned short int
INT_MIN	–32 768	Minimum value for an int
INT_MAX	32 767	Maximum value for an int
UINT_MAX	65 535	Maximum value for an unsigned int
LONG_MIN	–2 147 483 648	Minimum value for a long int
LONG_MAX	2 147 483 647	Maximum value for a long int
ULONG_MAX	4 294 967 295	Maximum value for an unsigned long int

Table 6–2. 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 (rounds to nearest integer)
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_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	–125	Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized float, double, or long double
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	128	Maximum integer such that FLT_RADIX raised to that power minus 1 is a representative finite float, double, or long double
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	1.19209290E–07F	Minimum positive float, double, or long double number *x*, such that $1.0 + x \neq 1.0$
FLT_MIN DBL_MIN LDBL_MIN	1.17549435E–38F	Minimum positive float, double, or long double
FLT_MAX DBL_MAX LDBL_MAX	3.40282347E+38F	Maximum float, double, or long double
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	–37	Minimum negative integer, 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 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.

6.8 math.h—Floating-Point Math

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.

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

The following table lists the floating-point math functions and indicates the page that contains more detailed information about the functions:

Function	Description	Page
acos	Returns the arc cosine of a value	6-34
asin	Returns the arc sine of a value	6-36
atan	Returns the arc tangent of a value	6-38
atan2	Returns the arc tangent of the values	6-38
ceil	Returns the smallest integer greater than or equal to a value	6-42
cos	Returns the cosine of a value	6-43
cosh	Returns the hyperbolic cosine of a value	6-43
exp	Returns the exponential function of a value	6-45
fabs	Returns the absolute value of a value	6-46
floor	Returns the largest integer less than or equal to a value	6-48
fmod	Returns the floating-point remainder of a value	6-48
frexp	Breaks the value into a normalized fraction and an integer power of 2	6-51
ldexp	Multiplies a value by an integer power of 2	6-55
log	Returns the natural logarithm of a value	6-56
log10	Returns the base-10 logarithm of a value	6-56
modf	Breaks a value into a signed integer and a signed fraction	6-61
pow	Returns the value raised to a specified power	6-62
sin	Returns the sine of a value	6-68
sinh	Returns the hyperbolic sine of a value	6-69
sqrt	Returns the nonnegative square root of a value	6-69
tan	Returns the tangent of a value	6-81
tanh	Returns the hyperbolic tangent of a value	6-81

6.9 setjmp.h—Bypass Normal Function Call and Return Conventions

The setjmp.h header defines a type, a macro, and a function for bypassing the normal function call and return discipline. The following table lists these definitions and indicates the page that contains more detailed information as applicable:

Definition	Description	Page
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	6-67
longjmp	A function that uses its jmp_buf argument to restore the program environment	6-67

6.10 stdarg.h—Variable Arguments

Some functions can have a varying number of arguments whose types can differ; such a function is called a *variable-argument function*. 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. A prototype must exist for a variable-argument function to correctly pass the function's arguments.

The following table lists the macros defined in stdarg.h and indicates the page that contains more detailed information about the macros:

Macro	Description	Page
va_arg	Accesses the next argument of type *type* in a variable-argument list	6-84
va_end	Resets the calling mechanism after using va_arg	6-84
va_start	Initializes ap to point to the first operand in the variable-argument list	6-84

6.11 stddef.h—Standard Definitions

The *stddef.h* header defines two types and two macros as described in the following table. These types and macros are used by several of the runtime-support functions.

Definition	Description
ptrdiff_t	A signed integer type that is the data type resulting from the subtraction of two pointers
size_t	An unsigned integer type that is the data type of the sizeof operator
NULL	Macro that expands to a null pointer constant (0)
OFFSETOF	Macro that 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).

6.12 stdio.h—I/O Functions

The stdio.h header defines types and macros and declares functions.

6.12.1 Type Declarations

The following table lists the types defined in stdio.h:

Type	Description
size_t	An unsigned integer type that is the data type of the sizeof operator
fpos_t	An unsigned long type that can uniquely specify every position within a file
FILE	A structure type to record all the information necessary to control a stream

6.12.2 Macro Declarations

The following table lists the macros defined in stdio.h:

Macro	Description
NULL	Expands to a null pointer constant(0). Originally defined in stddef.h. It is not redefined if it was already defined.
BUFSIZ	Expands to the size of the buffer that setbuf() uses
EOF	The end-of-file marker
FOPEN_MAX	Expands to the largest number of files that can be open at one time
FILENAME_MAX	Expands to the length of the longest file name in characters
L_tmpnam	Expands to the longest filename string that tmpnam() can generate
SEEK_CUR SEEK_SET SEEK_END	Expand to indicate the position (current, start-of-file, or end-of-file, respectively) in a file
TMP_MAX	Expands to the maximum number of unique filenames that tmpnam() can generate
stderr stdin stdout	Pointers to the standard error, input, and output files, respectively

6.12.3 Function Declarations

The following table lists the functions declared in stdio.h and indicates the page that contains more detailed information about the functions:

Function	Description	Page
add_device	Adds a device record to the device table	6-34
clearerr	Clears the EOF and error indicators for the stream pointed to	6-42
fclose	Flushes the stream pointed to and closes the file associated with that stream	6-46
feof	Tests the EOF indicator for the stream pointed to	6-46
ferror	Tests the error indicator for the stream pointed to	6-47
fflush	Flushes the I/O buffer for the stream pointed to	6-47
fgetc	Reads the next character in the stream pointed to	6-47
fgetpos	Stores the object pointed to to the current value of the file position indicator for the stream pointed to	6-47
*fgets	Reads the next size minus 1 characters from the stream pointed to into the specified array	6-48
*fopen	Opens the file pointed to	6-49
fprintf	Writes to the stream pointed to	6-49
fputc	Writes a single character to the specified stream	6-49
fputs	Writes the string to the specified stream	6-49
fread	Reads from the specified stream and stores the input to the specified array	6-50
*freopen	Opens the file pointed to	6-50
fscanf	Reads from the stream pointed to	6-51
fseek	Sets the file position indicator for the stream pointed to	6-51
fsetpos	Sets the file position indicator for the stream pointed to	6-52
ftell	Obtains the current value of the file position indicator for the stream pointed to	6-52
fwrite	Writes a block of data from the specified memory to the stream pointed to	6-52
getc	Reads the next character in the file pointed to	6-52
getchar	Reads the next character from the standard input device	6-53
*gets	Reads an input line from the standard input device	6-53
perror	Maps an error number to a string and prints the error message	6-61
printf	Writes to the standard output device	6-62
putc	Writes a character to the stream pointed to	6-62

Function	Description	Page
putchar	Writes a character to the standard output device	6-63
puts	Writes the string pointed to to the standard output device	6-63
remove	Makes the file with the name pointed to no longer available by that name	6-65
rename	Makes the file with the name pointed to known by the specified name	6-66
rewind	Sets the file position indicator for the stream pointed to	6-66
scanf	Reads the stream from the standard input device	6-66
setbuf	Specifies the buffer used by the stream pointed to	6-66
setvbuf	Defines and associates a buffer with a stream	6-68
sprintf	Writes to the array pointed to	6-69
sscanf	Reads from the string pointed to	6-70
*tmpfile	Creates a temporary file	6-82
*tmpnam	Generates a string that is a valid filename	6-82
ungetc	Writes a character to the stream pointed to	6-83
vfprintf	Writes to the stream pointed to	6-85
vprintf	Writes to the standard output device	6-85
vsprintf	Writes to the array pointed to	6-86

6.13 stdlib.h—General Functions

The stdlib.h header declares a macro, two types, and several functions.

6.13.1 Macro Declarations

The stdlib.h header declares the RAND_MAX macro. RAND_MAX is the maximum random number the rand function can return.

6.13.2 Type Declarations

The following table lists the types defined in stdlib.h:

Type	Description
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

6.13.3 Function Declarations

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

- Memory management functions that allow you to allocate and deallocate packets of memory. The amount of memory that these functions can use is defined by the symbol _SYSMEM_SIZE. By default, the amount of memory available for allocation is 1K words. You can change this amount at link time by invoking the linker with the –heap option and specifying the desired heap size as a constant directly after the option.
- Integer-arithmetic functions not provided as a standard part of the C language
- String conversion functions
- Searching and sorting functions
- Sequence-generation functions
- Program-exit functions

The following table lists the functions defined in stdlib.h:

Function	Description	Page
abort	Terminates a program abnormally	6-33
abs	Returns an absolute value	6-33

Function	Description	Page
atexit	Registers the function pointed to	6-39
atof	Converts a string to a floating-point value	6-39
atoi	Converts a string to an integer	6-39
atol	Converts a string to a long integer	6-39
*bsearch	Searches through an array for an object	6-40
*calloc	Allocates and clears memory	6-41
div	Performs integer division	6-44
exit	Terminates a program normally	6-45
free	Deallocates memory space allocated by malloc, calloc, or realloc	6-50
*getenv	Returns environment information associated	6-53
labs	Returns an absolute value	6-33
ldiv	Performs long integer division	6-44
ltoa	Converts a value to the equivalent string	6-57
*malloc	Allocates memory for an object	6-57
minit	Resets all the memory previously allocated by malloc, calloc, or realloc	6-59
qsort	Sorts an array	6-63
rand	Returns a sequence of pseudorandom integers to RAND_MAX	6-64
*realloc	Changes the size of memory pointed to	6-65
srand	Resets the random number generator	6-64
strtod	Converts a string to a floating-point value	6-79
strtol	Converts a string to a long integer	6-79
strtoul	Converts a string to an unsigned long integer	6-79

6.14 string.h—String Functions

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.

The string.h header declares the following functions:

Function	Description	Page
*memchr	Finds the first occurrence of a value	6-58
memcmp	Compares the characters in two strings	6-58
*memcpy	Copies characters from one string to another	6-58
*memmove	Moves characters from one string to another	6-59
*memset	Copies characters from one string to another	6-59
*strcat	Appends a string to the end of another string	6-70
*strchr	Finds the first occurrence of the specified character in a string	6-71
strcmp	Compares strings	6-71
strcoll	Compares strings	6-71
*strcpy	Copies a string	6-72
strcspn	Returns the length of the initial segment of a string	6-72
*strerror	Maps the error number in errno to an error message string	6-73
strlen	Returns the length of a string	6-74
*strncat	Appends characters from one string to another	6-74
strncmp	Compares strings	6-75
*strncpy	Copies characters from one string to another	6-76
*strpbrk	Locates the first occurrence of a character in a string	6-77
*strrchr	Finds the last occurrence of character in a string	6-77

Function	Description	Page
strspn	Returns the length of the initial segment of a string	6-78
*strstr	Finds the first occurrence of a string in another string	6-78
strtok	Breaks string into a series of tokens	6-80
strxfrm	Converts the characters of one string into another string	6-80

6.15 time.h—Time Functions

The *time.h* header declares a macro, several types, and functions that manipulate dates and times.

6.15.1 Macro Declarations

The time.h header declares the CLK_TCK macro. This macro is the number per second of the value returned by the clock function.

6.15.2 Type Declarations

The following table lists the types defined in time.h:

Type	Description
clock_t	An arithmetic type that represents time
time_t	An arithmetic type that represents time

tm	A structure that hold the components of calendar time, called *broken-down time*
	The tm 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 */`
	tm_isdst can have one of three values:
	❑ A positive value if daylight saving time is in effect
	❑ Zero if daylight saving time is not in effect
	❑ A negative value if the information is not available

6.15.3 Function Declarations

Note: Customizing Time Functions

All of the time functions depend on the clock and time functions, which you must customize for your system.

The functions deal with several types of time:

- Calendar time represents the current date (according to the Gregorian calendar) and time.
- Local time is the calendar time expressed for a specific time zone.
- Daylight saving time is a variation of local time.

Function	Description	Page
*asctime	Converts a time to a string	6-36
clock	Determines the processor time used	6-42
*ctime	Converts time to a string; expands inline if –x is used	6-43
difftime	Returns the difference between two calendar times	6-44
*gmtime	Converts local time to Greenwich Mean Time	6-54
*localtime	Converts time_t value to broken down time	6-56
mktime	Converts broken-down time to a time_t value	6-60
strftime	Formats a time into a character string	6-73
time	Returns the current calendar time	6-81

6.16 Description of Runtime-Support Functions and Macros

This section contains an alphabetical list of the functions and macros declared in the header files. The following table lists the functions and macros and directs you to the detailed description of the function or macro:

abort — *Abort*

Syntax

#included<stdlib.h>

void abort(void**);**

Defined in

exit.c in rts.src

Description

Terminates a program with an error code. The ’C54x implementation of the abort function calls the exit function with a value of 0 and is defined as follows:

```
void  abort ()
{
   exit(0);
}
```

A value of 0 makes the abort and exit functions equivalent. See the exit function on page 6-45.

abs/labs — *Absolute Value*

Syntax

#include <stdlib.h>

int abs(int val1**);**
long int labs(long int val2**);**

Defined in

abs.c in rts.src

Description

The following functions return the absolute value of an integer:

- The abs function returns the absolute value of an integer val1.
- The labs function returns the absolute value of a long integer val2.

Since int and long int are functionally equivalent types in ’C54x C, the abs and labs functions are also functionally equivalent. The function expands inline unless the –x0 compiler 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

Returns the arc cosine of a floating-point argument, x, which must be in the range [–1,1]. The return value is an angle in the range [0,π] radians.

Example

```
double realval, radians;

return (realval = 1.0;
radians = acos(realval);
return (radians);  /* acos return π/2 */
```

add_device — *Add Device to Device Table*

Syntax

#include <stdio.h>

int add_device(char *name,
unsigned flags**,**
int (* dopen**)(),**
int (* dclose**)(),**
int (* dread**)(),**
int (* dwrite**)(),**
int (* dlseek**)(),**
int (* dunlink**)(),**
int (* drename**)());**

Defined in

lowlev.c in rts.src

Description

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

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

Parameters

name	Character string denoting the device name
flags	Device characteristics. The flags denote the following: **_SSA** The device supports only one open stream at a time **_MSA** The device supports multiple open streams More flags can be added by defining them in stdio.h.
dopen, dclose, dread, dwrite, dlseek, dunlink, drename	Function pointers to the device drivers are called by the low-level functions to perform I/O on the specified device. You must declare these functions with the interface specified in subsection 6.6.3, *Overview of Low-Level I/O Implementation*, on page 6-10. The device drivers for the host that the 'C54x debugger is run on are included in the C I/O library.

Return Value

0 if successful
–1 if it fails

Example

This example does the following:

- Adds the device *mydevice* to the device table
- Opens a file named *test* on that device and associate it with the file **fid*
- Writes the string *Hello, world* into the file
- Closes the file

```
#include <stdio.h>

/*************************************************************************/
/* Declarations of the user–defined device drivers                       */
/*************************************************************************/
extern int my_open(char *path, unsigned flags, int fno);
extern int my_close(int fno);
extern int my_read(int fno, char *buffer, unsigned count);
extern int my_write(int fno, char *buffer, unsigned count);
extern int my_lseek(int fno, long offset, int origin);
extern int my_unlink(char *path);
extern int my_rename(char *old_name, char *new_name);

main()
{

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

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

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

   fclose(fid);
}
```

asctime — *Convert Internal Time to String*

Syntax

#include <time.h>

char *asctime(const struct tm *timeptr**);**

Defined in

asctime.c in rts.src

Description

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.

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, which must be in the range [–1,1]. The return value is an angle in the range [–π/2,π/2] radians.

Example

```
double realval, radians;
realval = 1.0;
radians = asin(realval);  /* asin returns π/2 */
```

assert — *Insert Diagnostic Information Macro*

Syntax

#include <assert.h>

void assert(int expression**);**

Defined in

assert.c in rts.src

Description

Tests an expression. Depending upon the value of the expression, assert does one of the following. This macro is useful for debugging.

- If expression is false, the assert macro writes information about the call that failed to the standard output device and aborts execution.
- If expression is true, the assert macro does nothing.

The header file that defines the assert macro refers to another macro, NDEBUG. If you have defined NDEBUG as a macro name when the assert.h header is included in the source file, the assert macro is defined 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.

Example

In this example, an integer, i, is divided by another integer j. Since dividing by 0 is an illegal operation, the example uses the assert macro to test j before the division. If j = = 0, assert issues a message and aborts the program.

```
int   i, j;
assert(j);
q = i/j;
```

atan — *Polar Arc Tangent*

Syntax

#include <math.h>

double atan(double x**);**

Defined in

atan.c in rts.src

Description

Returns the arc tangent of a floating-point argument, x. The return value is an angle in the range [–π/2,π/2] radians.

Example

```
double realval, radians;

realval = 0.0;
radians = atan(realval);     /* return value = 0 */
```

atan2 — *Cartesian Arc Tangent*

Syntax

#include <math.h>

double atan2(double y, double x**);**

Defined in

atan.c in rts.src

Description

Returns the inverse tangent of y/x. The function uses the signs of the arguments to determine the quadrant of the return value. Both arguments cannot be 0. The return value is an angle in the range [–π,π] radians.

Example

```
double rvalu, rvalv;
double radians;

rvalu  = 0.0;
rvalv  = 1.0;
radians = atan2(rvalr, rvalu);   /* return value = 0 */
```

atexit — *Register Function Called by Exit ()*

Syntax

#include <stdlib.h>

int atexit(void (*func)(void)**);**

Defined in

exit.c in rts.src

Description

Registers the function that is pointed to by func. The function is called without arguments at normal program termination. Registers up to 32 functions.

When the program exits through a call to the exit function, the functions that were registered are called without arguments in reverse order of their registration.

atof/atoi/atol — *Convert String to Number*

Syntax

#include <stdlib.h>

double atof(const char *nptr**);**
int atoi(const char *nptr**);**
long int atol(const char *nptr**);**

Defined in

atof.c, atoi.c, and atol.c in rts.src

Description

The following functions convert strings to numeric representations:

- The atof function converts a string into a floating-point value. The function expands inline if the –x compiler option is used. 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. The function expands inline if the –x compiler option is used. Argument nptr points to the string; the string must have the following format:

  [space] [sign] digits

The *space* is indicated by a space (character), a horizontal or vertical tab, a carriage return, a form feed, or a new line. Following the *space* is an optional *sign*, and the *digits* that represent the integer portion of the number. The fractional part of the number follows, then the exponent, including an optional *sign*.

The first character that cannot be part of the number terminates the string.

Since int and long are functionally equivalent in 'C54x 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(const void *key, const void *base, size_t nmemb,
size_t size**,** int (*compar)(const void *, const void *)**);**

Defined in

bsearch.c in rts.src

Description

Searches through an array of nmemb objects for a member that matches the object that key points to. Argument base points to the first member in the array; size specifies the size (in bytes) of each member.

The contents of the array must be in ascending order. If a match is found, the function returns a pointer to the matching member of the array; if no match is found, the function returns a null pointer (0).

Argument compar points to a function that compares key to the array elements. The comparison function must be declared as:

int cmp(const void *ptr1, const void *ptr2);

The cmp function compares the objects that ptr1 and ptr2 point to and returns one of the following values:

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

Example

```
#include <stdlib.h>
#include <stdio.h>

int list [] = {1, 3, 4, 6, 8, 9};
int idiff (const void *, const void *);

main( )
{
    int key = 8;
    int p = bsearch (&key, list, 6, 1, idiff);
        /* p points to list[4] */
}
int idiff (const void *il, const void *i2)
{
    return *(int *) i1 – *(int *) i2;
}
```

For an additional example, see strcmp on page 6-71.

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

Allocates size bytes (size is an unsigned integer or size_t) 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. The constant, _ _SYSMEM_SIZE, defines the size of the heap as 2K bytes. You can change this amount at link time by invoking the linker with the –heap option and specifying the desired size of the heap (in bytes) directly after the option. For more information, see subsection 5.1.4, *Dynamic Memory Allocation*, on page 5-5.

Example

This example uses the calloc routine to allocate and clear 20 bytes.

```
prt = calloc (10,2)  ;   /*Allocate and clear 20 bytes */
```

ceil — *Ceiling*

Syntax

#include <math.h>

double ceil(double x**);**

Defined in

ceil.c in rts.src

Description

Returns a floating-point number that represents the smallest integer greater than or equal to x. The function expands inline if the –x compiler option is used.

Example

```
extern double ceil();
double answer;
answer = ceil(3.1415);   /* answer = 4.0  */
answer = ceil(–3.5);     /* answer = –3.0 */
```

clearerr — *Clear EOF and Error Indicators*

Syntax

#include <stdio.h>

void clearerr(FILE *p**);**

Defined in

clearerr.c in rts.src

Description

Clears the EOF and error indicators for the stream that p points to.

clock — *Processor Time*

Syntax

#include <time.h>

clock_t clock(void**);**

Defined in

clock.c in rts.src

Description

Determines the amount of processor time used. It returns an approximation of the processor time used by a program since the program began running. The time in seconds divided by the value of the macro CLK_TCK is the return value.

If the processor time is not available or cannot be represented, the clock function returns the value of [(clock_t) –1]. For more information about the functions and types that the time.h header declares, see Section 6.15, *time.h—Time Functions*, on page 6-30.

Note: Writing Your Own Clock Function

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

cos — *Cosine*

Syntax

#include <math.h>

double cos(double x**);**

Defined in

sin.c in rts.src

Description

Returns the cosine of a floating-point number, x. The angle x is expressed in radians. An argument with a large magnitude might 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

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(const time_t *timer**);**

Defined in

ctime.c in rts.src

Description

Converts a calendar time (pointed to by timer) to local time in the form of a string. This is equivalent to:

```
asctime(localtime(timer))
```

The function returns the pointer returned by the asctime function. The function expands inline if the –x compiler option is used.

For more information about the functions and types that the time.h header declares, see Section 6.15, *time.h—Time Functions*, on page 6-30.

difftime — *Time Difference*

Syntax

#include <time.h>

double difftime(time_t time1, time_t time0**);**

Defined in

difftime.c in rts.src

Description

Calculates the difference between two calendar times, time1 minus time0. The return value expresses seconds.

For more information about the functions and types that the time.h header declares, see Section 6.15, *time.h—Time Functions*, on page 6-30.

div/ldiv — *Division*

Syntax

#include <stdlib.h>

div_t div(int numer, int denom**);**
ldiv_t ldiv(long numer, int denom**);**

Defined in

div.c in rts.src

Description

The following functions support integer division by returning numer (numerator) divided by denom (denominator). You can use these functions to get both the quotient and the remainder in a single operation.

- The div function performs integer division. The input arguments are integers; the function returns the quotient and the remainder in a structure of type div_t. The structure is defined as follows:

```
typedef struct
{
   int  quot;         /*  quotient    */
   int  rem;          /* remainder    */
} div_t;
```

- The ldiv function performs long integer division. The input arguments are long integers; the function returns the quotient and the remainder in a structure of type ldiv_t. The structure is defined as follows:

```
typedef struct
{
   long int  quot;    /*  quotient    */
   long int  rem;     /* remainder    */
} ldiv_t;
```

If the division produces a remainder, the sign of the quotient is the same as the algebraic quotient. The magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient.

Because ints and longs are equivalent types in ’C54x C, ldiv and div 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

Terminates a program normally. All functions registered by the atexit function are called in reverse order of their registration. The exit function can accept EXIT_FAILURE as a value. The exit function with a value of 0 is equivalent to the abort function. (See the abort function on page 6-33).

You can modify the exit function to perform application-specific shut-down tasks. The unmodified function simply runs in an infinite loop until the system is reset.

The exit function cannot return to its caller.

exp — *Exponential*

Syntax

#include <math.h>

double exp(double x**);**

Defined in

exp.c in rts.src

Description

Returns the exponential function of real number x. The return value is the number e raised to the power x. A range error occurs if the magnitude of x is too large.

Example

```
double x, y;

x = 2.0;
y = exp(x);        /* y = 7.38, which is e**2.0 */
```

fabs — *Absolute Value*

Syntax

#include <math.h>

double fabs(double x**);**

Defined in

fabs.c in rts.src

Description

Returns the absolute value of a floating-point number, x. The function expands inline unless the –x0 compiler option is used.

Example

```
double x, y;

x = –57.5;
y = fabs(x);       /* return value = +57.5 */
```

fclose — *Close File*

Syntax

#include <stdio.h>

int fclose(FILE *iop**);**

Defined in

fclose.c in rts.src

Description

Flushes the stream that iop points to. When the stream is flushed, the function closes the file associated with the stream.

feof — *Test EOF Indicator*

Syntax

#include <stdio.h>

int feof(FILE *p**);**

Defined in

feof.c in rts.src

Description

Tests the EOF indicator for a stream. The stream is pointed to by p.

ferror — *Test Error Indicator*

Syntax

#include <stdio.h>

int ferror(FILE *p**);**

Defined in

ferror.c in rts.src

Description

Tests the error indicator for a stream. The stream is pointed to by p.

fflush — *Flush I/O Buffer*

Syntax

#include <stdio.h>

int fflush(register **FILE** *iop**);**

Defined in

fflush.c in rts.src

Description

Flushes the I/O buffer for a stream. The stream is pointed to by iop.

fgetc — *Read Next Character*

Syntax

#include <stdio.h>

int fgetc(register **FILE** *fp**);**

Defined in

fgetc.c in rts.src

Description

Reads the next character in a stream. The stream is pointed to by fp.

fgetpos — *Store Object*

Syntax

#include <stdio.h>

int fgetpos(FILE *iop**,** fpos_t *pos**);**

Defined in

fgetpos.c in rts.src

Description

Stores the object pointed to by pos. The object is stored to the current value of the file position indicator for the stream pointed to by iop.

fgets — *Read Next Characters*

Syntax

#include <stdio.h>

char *fgets(char *ptr, register int size, register **FILE** *iop**);**

Defined in

fgets.c in rts.src

Description

Reads the specified number of characters from the stream pointed to by iop. The characters are placed in the array named by ptr. The number of characters read is size –1.

floor — *Floor*

Syntax

#include <math.h>

double floor(double x**);**

Defined in

floor.c in rts.src

Description

Returns a floating-point number. The number represents the largest integer less than or equal to x. The function expands inline if the –x compiler 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 floating-point remainder of x divided by y. If y == 0, the function returns 0. The function expands inline if the –x compiler option is used.

Example

```
double x, y, r;

x = 11.0;
y = 5.0;
r = fmod(x, y);       /* fmod returns 1.0 */
```

fopen — *Open File*

Syntax

#include <stdio.h>

FILE *fopen(const char *file, const char *mode**);**

Defined in

fopen.c in rts.src

Description

Opens the file that file points to. How to open the file is described by a string pointed to by mode.

fprintf — *Write Stream*

Syntax

#include <stdio.h>

int fprintf(FILE *iop, const char *format, ...**);**

Defined in

fprintf.c in rts.src

Description

Writes to the stream pointed to by iop. How to write the stream is described by a string pointed to by format.

fputc — *Write Character*

Syntax

#include <stdio.h>

int fputc(int c, register **FILE** *fp**);**

Defined in

fputc.c in rts.src

Description

Writes a character to a stream. The stream is pointed to by fp.

fputs — *Write String*

Syntax

#include <stdio.h>

int fputs(const char *ptr, register **FILE** *iop**);**

Defined in

fputs.c in rts.src

Description

Writes the string pointed to by ptr to a stream. The stream is pointed to by iop.

fread — *Read Stream*

Syntax

#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t count, **FILE** *iop**);**

Defined in

fread.c in rts.src

Description

Reads from the stream pointed to by iop. Stores the input in the array pointed to by ptr. The number of objects read is count. The size of the objects is size.

free — *Deallocate Memory*

Syntax

#include <stdlib.h>

void free(void *ptr**);**

Defined in

memory.c in rts.src

Description

Deallocates memory space (pointed to by ptr) that was previously allocated by a malloc, calloc, or realloc call. This makes the memory space available again. If you attempt to free unallocated space, the function takes no action and returns. For more information, see subsection 5.1.4, *Dynamic Memory Allocation*, on page 5-5.

Example

This example allocates ten bytes and frees them.

```
char *x;
x = malloc(10);       /*  allocate 10 bytes   */
free(x);              /*  free 10 bytes       */
```

freopen — *Open File*

Syntax

#include <stdio.h>

FILE *freopen(const char *file, const char *mode, register **FILE** *iop**);**

Defined in

fopen.c in rts.src

Description

Opens the file pointed to by file, and associates with it the stream pointed to by iop. How to open the file is described by a string pointed to by mode.

frexp — *Fraction and Exponent*

Syntax

#include <math.h>

double frexp(double value**,** int *exp**);**

Defined in

frexp.c in rts.src

Description

Breaks a floating-point number into a normalized fraction and the integer power of 2. The function returns a value with a magnitude in the range [1/2,1] or 0, so that value = = $x \times 2^{exp}$. The frexp function stores the power in the int pointed to by exp. If value is 0, both parts of the result are 0.

Example

```
double fraction;
int exp;

fraction = frexp(3.0, &exp);
/* after execution, fraction is .75 and exp is 2 */
```

fscanf — *Read Stream*

Syntax

#include <stdio.h>

int fscanf(FILE *iop, const char *format, ...**);**

Defined in

fscanf.c in rts.src

Description

Reads from the stream pointed to by iop. How to read the stream is described by a string pointed to by format.

fseek — *Set File Position Indicator*

Syntax

#include <stdio.h>

int fseek(register **FILE** *iop, long offset, int ptr**);**

Defined in

fseek.c in rts.src

Description

Sets the file position indicator for the stream pointed to by iop. The position is specified by ptr. For a binary file, use offset to position the indicator from ptr. For a text file, offset must be 0.

fsetpos — *Set File Position Indicator*

Syntax

#include <stdio.h>

int fsetpos(FILE *iop, const fpos_t *pos**);**

Defined in

fsetpos.c in rts.src

Description

Sets the file position indicator for the stream pointed to by iop to pos. The pointer pos must be a value from fgetpos() on the same stream.

ftell — *Get Current File Position Indicator*

Syntax

#include <stdio.h>

long ftell(FILE *iop**);**

Defined in

ftell.c in rts.src

Description

Gets the current value of the file position indicator for a stream. The stream is pointed to by iop.

fwrite — *Write Block of Data*

Syntax

#include <stdio.h>

size_t fwrite(const void *ptr, size_t size, size_t count, register **FILE** *iop**);**

Defined in

fwrite.c in rts.src

Description

Writes a block of data from the memory pointed to by ptr to a stream. The stream is pointed to by iop.

getc — *Read Next Character*

Syntax

#include <stdio.h>

int getc(FILE *p**);**

Defined in

fgetc.c in rts.src

Description

Reads the next character in a file. The file is pointed to by p.

getchar — *Read Next Character From Standard Input*

Syntax

#include <stdio.h>

int getchar(void);

Defined in

fgetc.c in rts.src

Description

Reads the next character from the standard input device.

getenv — *Get Environment Information*

Syntax

#include <stdlib.h>

char *getenv(const char *name**);**

Defined in

lddrv.c in rts.src

Description

Returns the environment information associated with name.

gets — *Read Next From Standard Input*

Syntax

#include <stdio.h>

char *gets(char *ptr**);**

Defined in

fgets.c in rts.src

Description

Reads an input line from the standard input device. The characters are placed in the array named by ptr.

gmtime — *Greenwich Mean Time*

Syntax

#include <time.h>

struct tm *gmtime(const time_t *timer**);**

Defined in

gmtime.c in rts.src

Description

The gmtime function converts a calendar time (pointed to by timer) into a broken-down time, which is expressed as Greenwich Mean Time (Coordinated Universal Time). For more information about the functions and types that the time.h header declares, see Section 6.15, *time.h—Time Functions*, on page 6-30.

isxxx — *Character Typing*

Syntax

#include <ctype.h>

int isalnum(int c**);**
int isalpha(int c**);**
int isascii(int c**);**
int iscntrl(int c**);**
int isdigit(int c**);**
int isgraph(int c**);**
int islower(int c**);**
int isprint(int c**);**
int ispunct(int c**);**
int isspace(int c**);**
int isupper(int c**);**
int isxdigit(int c**);**

Defined in

isxxx.c and ctype.c in rts.src
Also defined in ctype.h as macros

Description

Test a single argument, c, to see if it is a particular type of character — alphabetic, alphanumeric, numeric, ASCII, etc. If the test is true, the function returns a nonzero value; if the test is false, the function returns 0. The character typing functions include:

isalnum Identifies alphanumeric ASCII characters (tests for any character for which isalpha or isdigit is true)

isalpha Identifies alphabetic ASCII characters (tests for any character for which islower or isupper is true)

isascii Identifies ASCII characters (any character from 0–127)

iscntrl Identifies control characters (ASCII characters 0–31 and 127)

isdigit Identifies numeric characters between 0 and 9 (inclusive)

isgraph Identifies any nonspace character

islower	Identifies lowercase alphabetic ASCII characters
isprint	Identifies printable ASCII characters, including spaces (ASCII characters 32–126)
ispunct	Identifies ASCII punctuation characters
isspace	Identifies ASCII tab (horizontal or vertical), space bar, carriage return, form feed, and new line characters
isupper	Identifies uppercase ASCII alphabetic characters
isxdigit	Identifies hexadecimal digits (0–9, a–f, A–F)

The C compiler also supports a set of macros that perform these same functions. The macros have the same names as the functions but are prefixed with an underscore; for example, _isascii is the macro equivalent of the isascii function. In general, the macros execute more efficiently than the functions.

labs

See abs/labs on page 6-33.

ldexp

Multiply by a Power of Two

Syntax

#include <math.h>

double ldexp(double x**,** int exp**);**

Defined in

ldexp.c in rts.src

Description

Multiplies a floating-point number, x, by the power of 2 and returns $x \times 2^{exp}$. The exp can be a negative or a positive value. A range error occurs if the result is too large.

Example

```
double result;

result = ldexp(1.5, 5);         /* result is 48.0 */
result = ldexp(6.0, -3);        /* result is 0.75 */
```

ldiv

See div/ldiv on page 6-44.

localtime — *Local Time*

Syntax

#include <time.h>

struct tm *localtime(const time_t *timer**);**

Defined in

localtime.c in rts.src

Description

Converts a calendar time (pointed to by timer) into a broken-down time, which is expressed as local time. The function returns a pointer to the converted time.

For more information about the functions and types that the time.h header declares, see Section 6.15, *time.h—Time Functions*, on page 6-30.

log — *Natural Logarithm*

Syntax

#include <math.h>

double log(double x**);**

Defined in

log.c in rts.src

Description

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

Returns the base-10 logarithm of a real number, x. A domain error occurs if x is negative; a range error occurs if x is 0.

Example

```
float x, y;

x = 10.0;
y = log(x);        /* Return value = 1.0 */
```

longjmp

See setjmp/longjmp on page 6-67.

ltoa

Convert Long Integer to ASCII

Syntax

No prototype provided

int ltoa(long n, char *buffer**)**;

Defined in

ltoa.c in rts.src

Description

Converts a long integer, n, to an equivalent ASCII string and writes it into the buffer. 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.

The ltoa function is a nonstandard (non-ANSI) function provided for compatibility. The standard equivalent is sprintf. The function is not prototyped in rts.src.

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

Allocates space for an object of size bytes and returns a pointer to the space. If malloc cannot allocate the packet (that is, if it runs out of memory), it returns a null pointer (0). This function does not modify the memory it allocates.

The memory that malloc uses is in a special memory pool or heap. The constant _ _SYSMEM_SIZE defines the size of the heap as 2K bytes. You can change this amount at link time by invoking the linker with the –heap option and specifying the desired size of the heap (in bytes) directly after the option. For more information, see subsection 5.1.4, *Dynamic Memory Allocation*, on page 5-5.

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(const void *s, int c, size_t n**);**

Defined in

memchr.c in rts.src

Description

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 function expands inline if the –x compiler option is used.

The memchr function is similar to strchr, except that the object that memchr searches can contain values of 0 and c can be 0.

memcmp — *Memory Compare*

Syntax

#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n**);**

Defined in

memcmp.c in rts.src

Description

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 function expands inline if the –x compiler option is used.

memcpy — *Memory Block Copy—Nonoverlapping*

Syntax

#include <string.h>

void *memcpy(void *s1, const void *s2, size_t n**);**

Defined in

memcpy.c in rts.src

Description

Copies n characters from the object that s2 points to into the object that s1 points to. If you attempt to copy characters of overlapping objects, the function's behavior is undefined. The function returns the value of s1.

The memcpy function is similar to strncpy, except that the objects that memcpy copies can contain values of 0.

memmove — *Memory Block Copy—Overlapping*

Syntax

#include <string.h>

void *memmove(void *s1, const void *s2, size_t n**);**

Defined in

memmove.c in rts.src

Description

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 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

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 function expands inline if the –x compiler option is used.

minit — *Reset Dynamic Memory Pool*

Syntax

no prototype provided

void minit(void**);**

Defined in

memory.c in rts.src

Description

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 form the value specified by the –heap option. Default heap size is 1K words. For more information, see subsection 5.1.4, *Dynamic Memory Allocation*, on page 5-5.

mktime — *Convert to Calendar Time*

Syntax

#include <time.h>

time_t mktime(struct tm *timeptr**);**

Defined in

mktime.c in rts.src

Description

Converts a broken-down time, expressed as local time, into proper calendar time. The timeptr argument points to a structure that holds the broken-down time.

The function ignores the original values of tm_wday and tm_yday and does not restrict the other values in the structure. After successful completion of time conversions, tm_wday and tm_yday are set appropriately, and the other components in the structure have values within the restricted ranges. The final value of tm_mday is not sent until tm_mon and tm_year are determined.

The return value is encoded as a value of type time_t. If the calendar time cannot be represented, the function returns the value –1.

For more information about the functions and types that the time.h header declares and defines, see Section 6.15, *time.h—Time Functions*, on page 6-30.

Example

This example determines the day of the week that July 4, 2001, falls on.

```
#include <time.h>
static const char *const wday[] = {
            ″Sunday″, ″Monday″, ″Tuesday″, ″Wednesday″,
            ″Thursday″, ″Friday″, ″Saturday″ };

struct tm time_str;

time_str.tm_year  = 2001 – 1900;
time_str.tm_mon   = 7;
time_str.tm_mday  = 4;
time_str.tm_hour  = 0;
time_str.tm_min   = 0;
time_str.tm_sec   = 1;
time_str.tm_isdst = 1;

mktime(&time_str);

/*    After calling this function, time_str.tm_wday    */
/*    contains the day of the week for July 4, 2001    */
```

modf — *Signed Integer and Fraction*

Syntax

#include <math.h>

double modf(double x**, ***y**);**

Defined in

modf.c in rts.src

Description

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 x and stores the integer as a double at the object pointed to by y.

Example

```
double value, ipart, fpart;

value = –3.1415;
fpart = modf(value, &ipart);

/* After execution, ipart contains –3.0, */
/* and fpart contains –0.1415.           */
```

perror — *Map Error Number*

Syntax

#include <stdio.h>

void perror(const char *s**);**

Defined in

perror.c in rts.src

Description

Maps the error number in s to a string. The function then prints the error message.

pow — *Raise to a Power*

Syntax

#include <math.h>

double pow(double x, double y**);**

Defined in

pow.c in rts.src

Description

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

Example

```
double x, y, z;

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

printf — *Write to Standard Output*

Syntax

#include <stdio.h>

int printf(const char *format, ...**);**

Defined in

printf.c in rts.src

Description

Writes to the standard output device. How to write the stream is described by a string pointed to by format.

putc — *Write Character*

Syntax

#include <stdio.h>

int putc(int x, **FILE** *p**);**

Defined in

fputc.c in rts.src

Description

Writes a character to a stream. The stream is pointed to by p.

putchar — *Write Character to Standard Output*

Syntax

#include <stdlib.h>

int putchar(int x**);**

Defined in

fputc.c in rts.src

Description

Writes a character to the standard output device.

puts — *Write to Standard Output*

Syntax

#include <stdlib.h>

int puts(const char *ptr**);**

Defined in

fputs.c in rts.src

Description

Writes a string to the standard output device. The string is pointed to by ptr.

qsort — *Array Sort*

Syntax

#include <stdlib.h>

reentrant void qsort(void *base, size_t n, size_t size, int (*compar) ()**)**;

Defined in

qsort.c in rts.src

Description

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. Declare the comparison function as:

int cmp(const void *ptr1, const void *ptr2)

The cmp function compares the objects that ptr1 and ptr2 point to and returns one of the following values:

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

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 *i1, const void *i2)
{
   return *(int *)i1 – *(int *)i2;
}
```

For an additional example, see strcmp on page 6-71.

rand/srand *Random Integer*

Syntax

#include <stdlib.h>

int rand(void**);**
void srand(unsigned seed**)**;

Defined in

rand.c in rts.src

Description

The following functions work together to provide pseudorandom sequence generation:

- The rand function returns pseudorandom integers in the range 0–RAND_MAX. For the ’C54x C compiler, the value of RAND_MAX is 2 147 483 646 (2^{31} –2).
- The srand function sets the value of seed so that a subsequent call to the rand function produces a new sequence of pseudorandom numbers. The srand function does not return a value.

If you call rand before calling srand, rand generates the same sequence it would produce if you first called srand with a seed value of 1. If you call srand with the same seed value, rand generates the same sequence of numbers.

realloc — *Change Heap Size*

Syntax

#include <stdlib.h>

void *realloc(void *ptr, size_t size**);**

Defined in

memory.c in rts.src

Description

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 0.
- 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 the entire object must be moved to allocate more space, realloc returns a pointer to the new space. Any memory freed by this operation is deallocated. If an error occurs, the function returns a null pointer (0).

The memory that calloc uses is in a special memory pool or heap. The constant _ _SYSMEM_SIZE defines the size of the heap as 2K bytes. You can change this amount at link time by invoking the linker with the –heap option and specifying the desired size of the heap (in bytes) directly after the option. For more information, see subsection 5.1.4, *Dynamic Memory Allocation*, on page 5-5.

remove — *Remove File*

Syntax

#include <stdio.h>

int remove(const char *file**);**

Defined in

remove.c in rts.src

Description

Makes the file pointed to no longer available by that name. The file is pointed to by file.

rename — *Rename File*

Syntax

#include <stdio.h>

int rename(const char *old, const char *new**)**;

Defined in

lowlev.c in rts.src

Description

Renames the file pointed to by old. The new name is pointed to by new.

rewind — *Position File Position Indicator to Beginning of File*

Syntax

#include <stdio.h>

void rewind(register **FILE** *iop**)**;

Defined in

rewind.c in rts.src

Description

Sets the file position indicator for a stream to the beginning of the file. The file is pointed to by iop.

scanf — *Read Stream From Standard Input*

Syntax

#include <stdio.h>

int scanf(const char *format, ...**);**

Defined in

fscanf.c in rts.src

Description

Reads from the stream from the standard input device. How to read the stream is described by a string pointed to by format.

setbuf — *Specify Buffer for Stream*

Syntax

#include <stdio.h>

void setbuf(register **FILE** *iop, char *buf**);**

Defined in

setbuf.c in rts.src

Description

Specifies the buffer used by the stream pointed to by iop. If buf is set to null, buffering is turned off. No value is returned.

setjmp/longjmp *Nonlocal Jumps*

Syntax

#include <setjmp.h>

int setjmp(jmp_buf env**)**
void longjmp(jmp_buf env, int returnval**)**

Defined in

setjmp.c in rts.src

Description

The following functions enable bypassing the normal function call and return discipline:

- The setjmp function saves its calling environment in the jmp_buf argument for later use by the longjmp function.

  If the return is from a direct invocation, setjmp returns the value 0. If the return is from a call to the longjmp function, setjmp returns a nonzero value.

- The longjmp function restores the environment that was saved in the jmp_buf argument by the most recent invocation of setjmp. If setjmp was not invoked or if it terminated execution irregularly, the behavior of longjmp is undefined.

  After longjmp is completed, the program execution continues as if the corresponding invocation of setjmp had just returned the value specified by returnval. The longjmp function does not cause setjmp to return a value of 0 even if returnval is 0. If returnval is 0, setjmp returns the value 1.

The jmp_buf type is an array suitable for holding the information needed to restore a calling environment.

Example

These functions are typically used to effect an immediate return from a deeply nested function call:

```
#include <setjmp.h>

jmp_buf env;

main()
{
   int errcode;

   if ((errcode = setjmp(env)) == 0)
      nest1();
   else
      switch (errcode)
   . . .
}
   . . .
nest42()
{
   if (input() == ERRCODE42)
   /* return to setjmp call in main */
      longjmp (env, ERRCODE42);
   . . .
}
```

setvbuf — *Define and Associate Buffer With Stream*

Syntax

#include <stdio.h>

int setvbuf(register **FILE** *iop, register char *buf, register int type,
register size t_size**);**

Defined in

setbuf.c in rts.src

Description

Defines and associates the buffer used by the stream pointed to by iop. If buf is set to null, a buffer is allocated. If buf names a buffer, that buffer is used for the stream. The t_size parameter specifies the size of the buffer. The type parameter specifies the type of buffering as follows:

_IOFBF	Full buffering occurs
_IOLBF	Line buffering occurs
_IONBF	No buffering occurs

sin — *Sine*

Syntax

#include <math.h>

double sin(double x**);**

Defined in

sin.c in rts.src

Description

Returns the sine of a floating-point number, x. The angle x is expressed in radians. An argument with a large magnitude can produce a result with little or no significance.

Example

```
double radian, sval;     /* sval is returned by sin */

radian = 3.1415927;
sval = sin(radian);      /* –1 is returned by sin   */
```

sinh — *Hyperbolic Sine*

Syntax

#include <math.h>

double sinh(double x**);**

Defined in

sinh.c in rts.src

Description

Returns the hyperbolic sine of a floating-point number, x. A range error occurs if the magnitude of the argument is too large.

Example

```
double x, y;

x = 0.0;
y = sinh(x);       /* return value = 0.0 */
```

sprintf — *Write Stream*

Syntax

#include <stdio.h>

int sprintf(char *string, const char *format, ...**);**

Defined in

sprintf.c in rts.src

Description

Writes to the array pointed to by string. How to write the stream is described by a string pointed to by format.

sqrt — *Square Root*

Syntax

#include <math.h>

double sqrt(double x**);**

Defined in

sqrt.c in rts.src

Description

Returns the nonnegative square root of a real number, x. A domain error occurs if the argument is negative.

Example

```
double x, y;

x = 100.0;
y = sqrt(x);       /* return value = 10.0 */
```

srand — *See rand/srand on page 6-64.*

sscanf — *Read Stream*

Syntax

#include <stdio.h>

int sscanf(const char *str, const char *format, ...**);**

Defined in

sscanf.c in rts.src

Description

Reads from the string pointed to by str. How to read the stream is described by a string pointed to by format.

strcat — *Concatenate Strings*

Syntax

#include <string.h>

char *strcat(char *s1, const char *s2**);**

Defined in

strcat.c in rts.src

Description

Appends a copy of s2 (including a 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.

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(const char *s, int c**);**

Defined in

strchr.c in rts.src

Description

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).

Example

```
char *a = "When zz comes home, the search is on for zs.";
char *b;
char the_z = 'z';
b = strchr(a,the_z);
```

After this example, b points to the first z in zz.

strcmp/strcoll — *String Compare*

Syntax

#include <string.h>

int strcmp(const char *s1, const char *s2**);**
int strcoll(const char *s1, const char *s2**);**

Defined in

strcmp.c in rts.src

Description

Compares s2 with s1. The functions are equivalent. Both functions are supported to provide compatibility with ANSI C.

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 execute        */
   }
if (strcoll(stra, strc) == 0)
   {
      /* statements here execute also      */
   }
```

strcpy — *String Copy*

Syntax

#include <string.h>

char *strcpy(char *s1, const char *s2**);**

Defined in

strcpy.c in rts.src

Description

Copies s2 (including a terminating null character) into s1. If you attempt to copy strings that overlap, the function's behavior is undefined. The function returns a pointer to s1.

Example

In the following example, the strings pointed to by *a and *b are two separate and distinct memory locations. In the comments, the notation \0 represents the null character:

```
char *a = ”The quick black fox”;
char *b = ” jumps over ”;

/* a –> ”The quick black fox\0”   */
/* b –> ” jumps over \0”          */

strcpy(a,b);

/* a –> ” jumps over \0”          */
/* b –> ” jumps over \0”          */
```

strcspn — *Find Number of Unmatching Characters*

Syntax

#include <string.h>

size_t strcspn(const char *s1, const char *s2**);**

Defined in

strcspn.c in rts.src

Description

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 errno**);**

Defined in

strerror.c in rts.src

Description

The strerror function returns the string “string error”. This function is supplied to provide ANSI compatibility.

strftime — *Format Time*

Syntax

#include <time.h>

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

Defined in

strftime.c in rts.src

Description

Formats a time (pointed to by timeptr) according to a format string and returns the formatted time in the string s. Up to maxsize characters can be written to s. The format parameter is a string of characters that tells the strftime function how to format the time; the following list shows the valid characters and describes what each character expands to.

Character 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 equivalency of either *a.m.* or *p.m.*
- **%S** the *seconds* as a decimal number (00–50)

%U the *week* number of the year (Sunday is the first day of the week) as a decimal number (00–52)

%x the *date* representation

%X the *time* representation

%y the *year* without century as a decimal number (00–99)

%Y the *year* with century as a decimal number

%Z the *time zone* name, or by no characters if no time zone exists

For more information about the functions and types that the time.h header declares, see Section 6.15, *time.h—Time Functions*, on page 6-30.

strlen — *Find String Length*

Syntax

#include <string.h>

size_t strlen(const char *s**);**

Defined in

strlen.c in rts.src

Description

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.

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, const char *s2**,** size_t n**);**

Defined in

strncat.c in rts.src

Description

Appends up to n characters of s2 (including a terminating null character) to s1. The initial character of s2 overwrites the null character that originally terminated s1; strncat appends a null character to result. The function returns the value of s1. The function expands inline if the –x compiler option is used.

Example In the following example, the character strings pointed to by *a, *b, and *c were assigned the values shown in the comments. In the comments, the notation \0 represents the null character:

```
char *a, *b, *c;
size_t size = 13;
.
.
.
/* a--> "I do not like them,\0"                          */;
/* b--> " Sam I am, \0"                                  */;
/* c--> "I do not like green eggs and ham\0"             */;
strncat (a,b,size);
/* a--> "I do not like them, Sam I am, \0"               */;
/* b--> " Sam I am, \0"                                  */;
/* c--> "I do not like green eggs and ham\0"             */;
strncat (a,c,size);
/* a--> "I do not like them, Sam I am, I do not like\0"  */;
/* b--> " Sam I am, \0"                                  */;
/* c--> "I do not like green eggs and ham\0"             */;
```

strncmp *Compare Strings*

Syntax

#include <string.h>

int strncmp(const char *s1, const char *s2, size_t n**);**

Defined in strncmp.c in rts.src

Description 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 execute       */
   }
if (strcomp(stra, strc, size) == 0)
   {
      /* statements here execute also  */
   }
```

strncpy — *String Copy*

Syntax

#include <string.h>

char *strncpy(char *s1, const char *s2**,** size_t n**);**

Defined in

strncpy.c in rts.src

Description

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. The function expands inline if the –x compiler option is used.

Example

Note that strb contains a leading space to make it five characters long. Also note that the first five characters of strc are an *I*, a space, the word *am*, and another space, so that after the second execution of strncpy, stra begins with the phrase *I am* followed by two spaces. In the comments, the notation \0 represents the null character.

```
char *stra = ″she's the one mother warned you of″;
char *strb = ″ he's″;
char *strc = ″I am the one father warned you of″;
char *strd = ″oops″;
int length = 5;
```

```
strncpy (stra,strb,length);

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

```
strncpy (stra,strc,length);

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

```
strncpy (stra,strd,length);

/* stra--> ″oops\0″                               */;
/* strb--> ″ he's\0″                              */;
/* strc--> ″I am the one father warned you of\0″  */;
/* strd--> ″oops\0″                               */;
```

strpbrk — *Find Any Matching Character*

Syntax

#include <string.h>

char *strpbrk(const char *s1, const char *s2**);**

Defined in

strpbrk.c in rts.src

Description

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(const char *s**,** int c**);**

Defined in

strrchr.c in rts.src

Description

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 function expands inline if the –x compiler option is used.

Example

```
char *a = ”When zz comes home, the search is on for zs”;
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(const char *s1, const char *s2**);**

Defined in

strspn.c in rts.src

Description

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 = strspn(stra,strb);     /* length = 3 */
length = strspn(stra,strc);     /* length = 0 */
```

strstr — *Find Matching String*

Syntax

#include <string.h>

char *strstr(const char *s1, const char *s2**);**

Defined in

strstr.c in rts.src

Description

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

String to Number

Syntax

#include <stdlib.h>

double strtod(const char *nptr**,** char **endptr**);**
long int strtol(const char *nptr**,** char **endptr, int base**);**
unsigned long strtoul(const char *nptr, char **endptr, int base**);**

Defined in

strtod.c, strtol.c, and strtoul in rts.src

Description

The following 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 what base to interpret the string in.

- The strtod function converts a string to a floating-point value. The string must have the following format:

  [space] [sign] digits [.digits] [e|E [sign] integer]

  The function returns the converted string; if the original string is empty or does not have the correct format, the function returns a 0. If the converted string would cause an overflow, the function returns ±HUGE_VAL; if the converted string would cause an underflow, the function returns 0. If the converted string overflows or underflows, the global errno variable 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. Specify the string in the following format:

  [space] [sign] digits [.digits] [e|E [sign] integer]

The space is indicated by a horizontal or vertical tab, space bar, carriage return, form feed, or new line. Following the space is an optional sign and digits that represent the integer portion of the number. The fractional part of the number follows, then the exponent, including an optional sign.

The first unrecognized character terminates the string. The pointer that endptr points to is set to point to this character.

strtok — *Break String into Token*

Syntax

#include <string.h>

char *strtok(char *s1, const char *s2**);**

Defined in

strtok.c in rts.src

Description

Successive calls 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\0 because strtok has inserted a null character where the first space used to be. In the comments, the notation \0 represents the null character.

```
char *stra = ”excuse me while I kiss the sky”;
char *ptr;

ptr = strtok (stra,” ”); /* ptr –> ”excuse\0” */
ptr = strtok (0,” ”);    /* ptr –> ”me\0”     */
ptr = strtok (0,” ”);    /* ptr –> ”while\0”  */
```

strxfrm — *Convert Characters*

Syntax

#include <string.h>

size_t strxfrm(char *tostring, const char *fromstring, size_t n**);**

Defined in

strxfrm.c in rts.src

Description

Converts the characters of one string into another string. The n number of characters pointed to by fromstring are converted into the n characters pointed to by tostring.

tan — *Tangent*

Syntax

#include <math.h>

double tan(double x**);**

Defined in

tan.c in rts.src

Description

Returns the tangent of a floating-point number, x. The angle x is expressed in radians. An argument with a large magnitude can produce a result with little or no significance.

Example

```
double x, y;

x = 3.1415927/4.0;
y = tan(x);           /* return value = 1.0 */
```

tanh — *Hyperbolic Tangent*

Syntax

#include <math.h>

double tanh(double x**);**

Defined in

.tanhc in rts.src

Description

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

Determines the current calendar time, represented in seconds. If the calendar time is not available, the function returns –1. If timer is not a null pointer, the function also assigns the return value to the object that timer points to.

For more information about the functions and types that the time.h header declares, see Section 6.15, *time.h—Time Functions*, on page 6-30.

Note: The time Function Is Target-System Specific

The time function is target-system specific, so you must write your own time function.

tmpfile — *Create Temporary File*

Syntax

```
#include <stdlib.h>
FILE *tmpfile(void);
```

Defined in tmpfile.c in rts.src

Description Creates a temporary file.

tmpnam — *Generate Valid Filename*

Syntax

```
#include <stdlib.h>
char *tmpnam(char *s);
```

Defined in tmpnam.c in rts.src

Description Generates a string that is a valid filename.

toascii — *Convert to ASCII*

Syntax

```
#include <ctype.h>
int toascii(int c);
```

Defined in toascii.c in rts.src

Description Ensures that c is a valid ASCII character by masking the lower seven bits. There is also an equivalent macro call _toascii.

tolower/toupper *Convert Case*

Syntax

#include <ctype.h>

int tolower(int c**);**
int toupper(int c**);**

Defined in

tolower.c and toupper.c in rts.src

Description

The following functions convert the case of a single alphabetic character, c, into uppercase or lowercase:

- The tolower function converts an uppercase argument to lowercase. If c is already in lowercase, tolower returns it unchanged.
- The toupper function converts a lowercase argument to uppercase. If c is already in uppercase, toupper returns it unchanged.

The functions have macro equivalents named _tolower and _toupper.

Example

```
tolower ('A')        /* returns 'a' */
tolower ('+')        /* returns '+' */
```

ungetc *Write Character to Stream*

Syntax

#include <stdio.h>

int ungetc(int c, **FILE** *p**);**

Defined in

ungetc.c in rts.src

Description

Writes the character c to a stream. The stream is pointed to by p.

va_arg/va_end/ va_start

Variable-Argument Macros

Syntax

```
#include <stdarg.h>
typedef *va_list;
va_arg(ap, type);
void va_end(ap);
void va_start(ap, parmN);
va_list *ap
```

Defined in

stdarg.h

Description

The following macros step through a function's argument list at runtime.

Some functions are called with a varying number of arguments that have varying types. Such a function, called a *variable-argument function*, can use these macros to step through its argument list at runtime. The ap parameter points to an argument in the variable-argument list.

- The va_start macro initializes ap to point to the first argument in an argument list for the variable-argument function. The parmN parameter points to the right-most parameter in the fixed, declared list.
- The va_arg macro returns the value of the next argument in a call to a variable-argument function. Each time you call va_arg, it modifies ap so that successive arguments for the variable-argument function can be returned by successive calls to va_arg (va_arg modifies ap to point to the next argument in the list). The type parameter is a type name; it is the type of the current argument in the list.
- The va_end macro resets the stack environment after va_start and va_arg are used.

You must call va_start to initialize ap before calling va_arg or va_end.

Example

```
int    printf (char *fmt...);
       va_list ap;
       va_start(ap, fmt);
       .
       .
       .
       i = va_arg(ap, int);     /* Get next arg, an integer */
       s = va_arg(ap, char *);  /* Get next arg, a string   */
       l = va_arg(ap, long);    /* Get next arg, a long     */
       .
       .
       .
       va_end(ap);              /* Reset                    */
}
```

vfprintf — *Write to Stream*

Syntax

#include <stdio.h>

int vfprintf(FILE *iop, const char *format, va_list ap**);**

Defined in

vfprintf.c in rts.src

Description

Writes to the stream pointed to by iop. How to write the stream is described by a string pointed to by format. The ap parameter is the argument list.

vprintf — *Write to Standard Output*

Syntax

#include <stdio.h>

int vprintf(const char *format, va_list ap**);**

Defined in

vprintf.c in rts.src

Description

Writes to the standard output device. How to write the stream is described by a string pointed to by format. The ap parameter is the argument list.

vsprintf — *Write Stream*

Syntax

```
#include    <stdio.h>

int vsprintf(char *string, const char *format, va_list ap);
```

Defined in

vsprintf.c in rts.src

Description

Writes to the array pointed to by string. How to write the stream is described by a string pointed to by format. The ap parameter is the argument list.

Chapter 7

Library-Build Utility

When using the C compiler, you can compile your code under a number of different configurations and options that are not necessarily compatible with one another. Since it would be cumbersome to include all possible combinations in individual runtime-support libraries, this package includes the source file, rts.src, that contains all runtime-support functions. You can custom build your own runtime-support libraries using selected options by using the mk500 utility described in this chapter and the archiver described in the *TMS320C54x Assembly Language Tools User's Guide*.

You install the mk500 utility on your host according to your system's conventions by using the procedure described in the *TMS320C54x Code Generation Tools Getting Started Guide*. All tools must be placed in your PATH statement. The utility ignores and disables the environment variables TMP, C_OPTION, and C_DIR.

Topic **Page**

7.1 Invoking the Library-Build Utility

The general syntax for invoking the library utility is:

mk500 [*options*] *src_arch1* [**–l***obj.lib1*] [*src_arch2* [**–l***obj.lib2*]] ...

mk500 is the command that invokes the utility.

options can appear anywhere on the command line or in a linker command file. (Options are discussed below and in Section 7.2, *Options Summary.*)

src_arch is the name of a source archive file. For each source archive named, mk500 builds 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, mk500 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.

Library-build 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, assembler, and linker. 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 mk500 to use header files contained in the source archive and leave them in the current directory after the utility has completed execution. You can use this option to install the runtime-support header files from the rts.src archive that is shipped with the tools.

––k instructs mk500 to overwrite files. By default, the utility aborts any time it attempts to create an object file when an object file of the same name already exists in the current directory, regardless of whether you specified the name or the utility derived it.

––q instructs mk500 to suppress header information (quiet).

––u instructs mk500 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).

7.2 Options Summary

The other options that can be used with the library-build utility correspond directly to the options that the compiler uses. These options are described in detail in subsection 2.1.3, *Changing Compiler Behavior With Compiler Options*, on page 2-5. The following table provides a summary of the options that you can use with the utility.

General Options	**Effect**
–g	Enable symbolic debugging
–r*register*	reserve global register
Parser Options	**Effect**
–p?	Enable trigraph expansion
–pk	Make code K&R compatible
–pw	Suppress warning messages
Optimizer Options	**Effect**
–o0	Compile with optimization register optimization
–o1	Compile with optimization + local optimization
–o2 (or –o)	Compile with optimization + global optimization
–o3	Compile with optimization + file optimization Note that mk500 automatically sets –oI0 and –op0.
–ox (equivalent to –x2)	Define _INLINE + above + invoke optimizer (at –o2 if not specified differently)
Inlining Options	**Effect**
–x1	Enable intrinsic function inlining
–x2 (or –x)	Define _INLINE + above + invoke optimizer (at –o2 if not specified differently)

Runtime Model Options	Effect
–ma	Assume aliased variables
–mb	Avoid RPT for structure moves
–mf	All calls are far calls and all returns will be far returns
–mn	Enable optimization disabled by –g
–ms	Optimize for space instead of for speed
–mx	Avoid ’C54x first run silicon bugs
Type Checking Options	**Effect**
–tf	Relax prototype checking
–tp	Relax pointer combination checking
Assembler Options	**Effect**
–as	Keep labels as symbols
Default File Extension Options	**Effect**
–ea<*.ext*>	Extension for assembly files (default is .asm)
–eo<*.ext*>	Extension for object files (default is .obj)

Appendix A

Optimizations

The TMS320C54x 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, see Section 2.3, *Using the C Compiler Optimizer*, on page 2-36.) However, the code generator performs some optimizations, particularly the TMS320C54x-specific optimizations, which you cannot selectively enable or disable.

Topic **Page**

A.1 Optimizations Performed

This appendix describes two categories of optimizations: ’C54x-specific and general. ’C54x-specific optimizations are designed especially for ’C54x architecture. General optimizations improve any C code. Both types of optimizations are performed throughout the compiler.

’C54x-Specific Optimizations

- Cost-based register allocation
- Autoincrement addressing
- Repeat blocks
- Delays, branches, calls, and returns

General Optimizations

- Algebraic reordering, symbolic simplification, constant folding
- Alias disambiguation
- Copy propagation
- Data flow optimizations
- Branch optimizations, control-flow simplification
- Loop induction variable optimizations, strength reduction
- Loop rotation
- Loop invariant code motion
- Inline expansion of runtime-support library functions

A.2 TMS320C54x-Specific Optimizations

TMS320C54x-specific optimizations are designed especially for ’C54x architecture. These optimizations do not apply to general C code since they work on the sections of code that apply only to the ’C54x device.

A.2.1 Cost-Based Register Allocation

The optimizer, when enabled, allocates registers to user variables and compiler temporary values according to their type, use, and frequency. Variables used within loops are weighted to have priority over others, and those variables whose uses do not overlap may be allocated to the same register.

A.2.2 Autoincrement Addressing

For pointer expressions of the form *p++, the compiler uses efficient TMS320C54x 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–1 on page A-4.

A.2.3 Repeat Blocks

The ’C54x supports zero-overhead loops with the RPTB (repeat block) instruction. With the optimizer, the compiler can detect loops controlled by counters and generate them using the efficient repeat forms. The iteration count can be either a constant or an expression.

Example A–1. Repeat Blocks, Autoincrement Addressing, Parallel Instructions, Strength Reduction, Induction Variable Elimination, Register Variables, and Loop Test Replacement

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

TMS320C54x C Compiler Output:

```
_scale:
          FRAME    #–2
          SSBX     SXM
          ST#      0,*SP(1)
          STL      A,*SP(0)
          LD       *SP(1),A
          SUB      #10,A,A
          BCL      3,AGEQ              ; branch occurs
L2:
          . . .
          MVDK     *SP(1),*(AR1)
          STL      B,*AR1(_a)
          ADDM     #1,*SP(1)
          LD       *SP(1),A
          SUB      #10,A,A
          BC       L2,ALT              ; branch occurs
L3:
          . . .
```

Induction variable elimination and loop test replacement allow the compiler to recognize the loop as a simple counting loop and then generate a repeat block. Strength reduction turns the array references into efficient pointer autoincrements.

A.2.4 Delays, Branches, Calls, and Returns

The ’C54x provides a number of delayed branch, call, and return instructions. Three of these are used by the compiler: branch unconditional (BD), call to a named function (CALLD), and simple return (RETD). These instructions execute in two fewer cycles than their nondelayed counterparts. They execute two instruction words after they enter the instruction stream. Sometimes it is necessary to insert a NOP instruction after a delayed instruction to ensure proper operation of the sequence. This is one word of code longer than a nondelayed sequence, but it is still one cycle faster. Note that the compiler inserts a comment in the instruction sequence where the delayed instruction executes. See Example A–2 on page A-6.

Example A–2. Delayed Branch, Call, and Return Instructions

```
main()
{
    int i0, i1;

    while (input(&i0) && input(&i1))
      process(i0, i1);
}
```

TMS320C54x C Compiler Output:

```
_main:
      SAR    AR0,*+           ; function prolog
      POPD   *+               ; save AR0 and return address
      SAR    AR1,*            ; begin to set up local frame
      BD     L2               ; begin branch to loop control
      LARK   AR0,3            ; finish setting up local frame
      LAR    AR0,*0+

***   B      L2 OCCURS        ; branch to loop control
L1:                           ; loop body
      LARK   AR2,2            ; AR2 = &i1
      MAR    *0+
      LAC    *–,AR1           ; ACC = *AR2, AR2 = &i0
      SACL   *+,AR2           ; stack ACC
      CALLD  _process         ; begin call
      LAC    * ,AR1           ; ACC = *AR2
      SACL   *+               ; stack ACC
***   CALL   _process OCCURS  ; call occurs
      SBRK   2                ; pop stack
L2:                           ; loop control
      MAR    * ,AR5           ; AR5 = &i0
      LARK   AR5,1
      CALLD  _input           ; begin call
      MAR    *0+,AR1
      SAR    AR5,*+           ; stack AR5
***   CALL   _input OCCURS    ; call occurs
      MAR    *–               ; clear stack
      BZ     EPI0_1           ; quit if _input returns 0
      MAR    * ,AR4           ; AR4 = &i1
      LARK   AR4,2
      CALLD  _input           ; begin call
      MAR    *0+,AR1
      SAR    AR4,*+           ; stack AR4
***   CALL   _input OCCURS    ; call occurs
      MAR    *–,AR2           ; clear stack
      BNZ    L1               ; continue if _input returns !0
EPI0_1:
      MAR    * ,AR1           ; function epilog
      SBRK   4                ; clear local frame
      PSHD   *–               ; push return address on hardware stack
      RETD                    ; begin return
      LAR    AR0,*            ; restore AR0
      NOP                     ; necessary, no PSHD in delay slot
***   RET OCCURS              ; return occurs
          ...
```

A.3 General Optimizations

General optimizations improve any C code. These optimizations apply to all C code since they do not take into account the ’C54x device architecture.

A.3.1 Algebraic Reordering/Symbolic Simplification/Constant Folding

For optimal evaluation, the compiler simplifies expressions into equivalent forms requiring fewer instructions or registers. For example, the expression (a + b) – (c + d) takes six instructions to evaluate; it can be optimized to ((a + b) – c) – d, which takes only four instructions. Operations between constants are folded into single constants. For example, a = (b + 4) – (c + 1) becomes a = b – c + 3. See Example A–3 on page A-9.

A.3.2 Alias Disambiguation

Programs written in the C language generally use many pointer variables, such as symbols, pointer references, and structure references. These pointer variables are referred to as *l values* (lowercase L).

Compilers are often unable to determine whether two or more l values refer to the same memory location. This prevents the compiler from retaining values in registers because it cannot be sure that the register and memory continue to hold the same values over time.

Alias disambiguation is a technique that determines when two pointer expressions cannot point to the same location, allowing the compiler to freely optimize such expressions.

A.3.3 Data-Flow Optimizations

Collectively, the following data-flow optimizations replace expressions with more efficient 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).

- **Copy Propagation**

  Following an assignment to a variable, the compiler replaces references to the variable with its value. The value can be another variable, a constant, or a common subexpression. This can result in increased opportunities for constant folding, common subexpression elimination, or even total elimination of the variable. See Example A–3 on page A-9 and Example A–4 on page A-11.

- **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–3 on page A-9.

- **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–3 on page A-9.

Example A–3. 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);
      ...
   }
```

TMS320C54x C Compiler Output:

```
_simp:
    . . .
*****************************************************
* b = j * 5;
*****************************************************
    LD     #(-3+LF1), A    ; A = j
    MPY    #5, A           ; A = j * 5
    STL    A, 4-LF1        ; store *(A) at b
*****************************************************
* call(3, b, j << 3, (j >> 3) + (j << b));
*****************************************************
    LT     *               ; t = *AR2 (b)
    SBRK   4-LF1           ; AR2 = &j
    LACT   * ,AR1          ; ACC = j << b
    SACL   * ,AR2          ; save off ACC on TOS (top of stack)
    SSXM                   ; need sign extension for right shift
    LAC    * ,12,AR1       ; high ACC = j >> 3
    ADD    * ,15           ; add TOS to high ACC
    SACH   *+,1,AR2        ; stack high ACC
    LAC    * ,3,AR1        ; ACC = j << 3
    SACL   *+,AR2          ; stack ACC
    ADRK   4-LF1            ; AR2 = &b
    LAC    * ,AR1          ; ACC = b
    SACL   *+              ; stack ACC
    CALLD  _call           ; call begins
    LACK   3               ; ACC = 3
    SACL   *+              ; stack ACC
***  CALL   _call OCCUR     ; call occurs
    . . .
```

The constant 3, assigned to a, is copy-propagated to all uses of a; a becomes a dead variable and is eliminated. The sum of multiplying j by 3 (a) and 2 is simplified into b = j * 5, which is recognized as a common subexpression. The assignments to c and d are dead and are replaced with their expressions. These optimizations are performed across jumps.

A.3.4 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. Each case in a switch statement is 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–4 on page A-11.

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

Example A–4. 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;
         }
   }
```

TMS320C54x C Compiler Output:

```
_fsm:
   . . .
*
* AR5   assigned to variable 'input'
*
       LAC    *+        ; initial state == ALPHA
       BNZ    L5        ; if (input != 0) goto state GAMMA
L2:
       LAC    *+        ; state == BETA
       BZ     L4        ; if (input == 0) goto state GAMMA
       LAC    *+        ; state == ALPHA
       BZ     L2        ; if (input == 0) goto state BETA
       B      L5        ; else    goto state GAMMA
L4:
       LAC    *+        ; state == GAMMA
       BNZ    EPI0_1    ; if (input != 0) goto state OMEGA
L5:
       LARP   AR5
L6:
       LAC    *+        ; state = GAMMA
       BZ     L6        ; if (input == 0) goto state GAMMA
EPI0_1:                 ; state == OMEGA
       . . .
```

A.3.5 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 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–1 on page A-4 and Example A–5 on page A-13.

A.3.6 Loop Rotation

The compiler evaluates loop condition statements 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.

A.3.7 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–5 on page A-13.

A.3.8 Inline Expansion of Runtime-Support Library Functions

The compiler replaces calls to small functions with inline code, saving the overhead associated with a function call as well as providing increased opportunities to apply other optimizations. See Example A–5 on page A-13.

Example A–5. Inline Function Expansion

```
#include <string.h>
struct s { int a,b,c[10]; };
struct t { int x,y,z[10]; };

proc_str(struct s *ps, struct t *pt)
{
   . . .
   memcpy(ps,pt,sizeof(*ps));
   . . .
}
_proc_str:
   . . .
```

TMS320C54x C Compiler Output:

```
*
* AR5 assigned to variable 'memcpy_1_rfrom'
* AR6 assigned to variable 'memcpy_1_rto'
* BRCR assigned to temp variable 'L$1'
*
      . . .

      LARK   AR2,-3+LF1 ; AR2 = &ps
      MAR    *0+
      LAR    AR6,*-     ; AR6 = ps, AR2 = &pt
      LAR    AR5,* ,AR5 ; AR5 = pt
      LACK   11
      SAMM   BRCR       ; repeat 12 times
      RPTB   L4-1
      LAC    *+,AR6     ; *ps++ = *pt++
      SACL   *+,AR5
      NOP               ; must have 3 words in repeat block
L4:
      . . .
```

The compiler finds the intermediate file code for the C function memcpy() in the inline library and copies it in place of the call. Note the creation of variables memcpy_1_from and memcpy_1_to, corresponding to the parameters of memcpy. (Often, copy propagation can eliminate such assignments to parameters of inlined functions when the arguments are not referenced after the call.)

Appendix B

Invoking the Compiler Tools Individually

The TMS320C54x C compiler offers you the versatility of invoking all of the tools at once, using the shell, or invoking each tool individually. To satisfy a variety of applications, you can invoke the compiler (parser and code generator), the assembler, and the linker as individual programs. This section also describes how to invoke the interlist utility outside the shell.

Topic **Page**

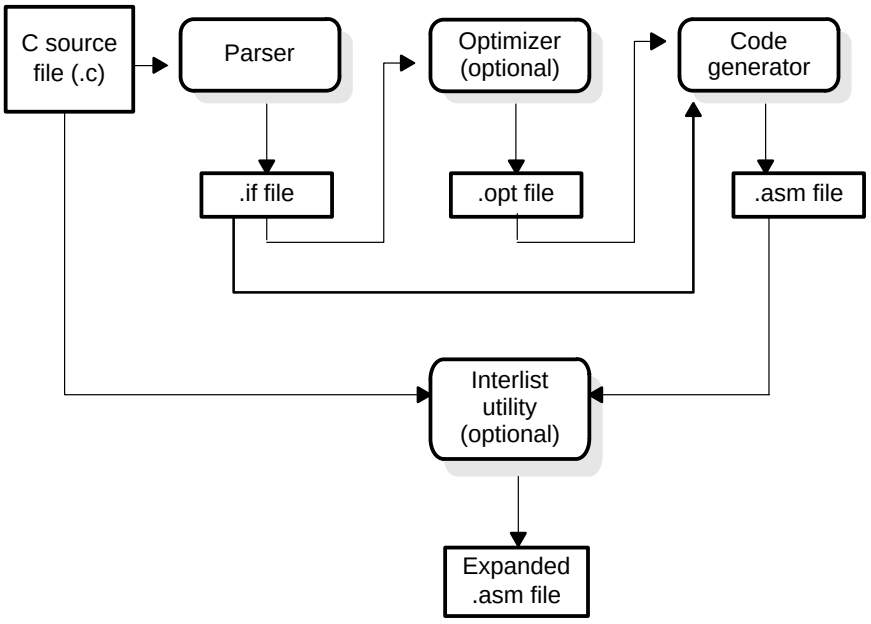
B.1 Which Tools Can be Invoked Individually

To satisfy a variety of applications, you can invoke the compiler, the assembler, and the linker as individual programs.

B.1.1 About the Compiler

The compiler is made up of three distinct programs: the parser, the optimizer, and the code generator. The compiler can also invoke the interlist utility.

Figure B–1. Compiler Overview



B.1.1.1 About the Parser

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. Section B.2, *Invoking the Parser Individually*, on page B-4 describes how to run the parser. The parser, in addition, can be run in two passes: the first pass pre-processes the code, and the second pass parses the code.

B.1.1.2 About the Optimizer

The optimizer is an optional pass that runs between the parser and the code generator. The input is the intermediate file (.if) produced by the parser. When you run the optimizer, you choose the level of optimization. The optimizer performs the optimizations on the intermediate file and produces a highly efficient version of the file in the same intermediate file format. Section 2.3, *Using the C Compiler Optimizer*, on page 2-36 describes the optimizer.

B.1.1.3 About the Code Generator

The input for the code generator is the intermediate file produced by the parser (.if) or the optimizer (.opt). The code generator produces an assembly language source file. Section B.4, *Invoking the Code Generator*, on page B-9 describes how to run the code generator.

B.1.1.4 About the 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 2.5, *Using the Interlist Utility*, on page 2-45 and Section B.5, *Invoking the Interlist Utility*, on page B-11 describe the use of the interlist utility.

B.1.2 About the Assembler and Linker

The input for the **assembler** is the assembly language file produced by the code generator. The assembler produces a COFF object file. The assembler is described fully in the *TMS320C54x Assembly Language Tools User's Guide.*

The input for the **linker** is the COFF object file produced by the assembler. The linker produces an executable object file. Chapter 3, *Linking C Code*, describes how to run the linker. The linker is described fully in the *TMS320C54x Assembly Language Tools User's Guide.*

B.2 Invoking the Parser Individually

The first step in compiling a ’C54x C program is to invoke the C parser. The parser reads the source file, performs preprocessing functions, checks syntax, and produces an intermediate file that can be used as input for the code generator or the optimizer. To invoke the parser, enter the following:

ac500 *input file* [*output file*] [*options*]

ac500 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*. If you don’t specify an input file, the parser prompts you for one.

output file names the intermediate file that the parser creates. If you 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 B–1 shows the parser options, the shell options, and the corresponding functions.

Note: Using Wildcards

You cannot use wildcards if you invoke the parser individually.

Table B–1. Parser Options

Parser Option	Function	See Shell Option	See Page
–d*name* [=def]	Predefines macro *name*	–d*name* [=def]	2-14
–e	Treats code-E errors as warnings	–pe	2-18, 2-49
–i*dir*	Defines #include search path	–i*dir*	2-14, 2-32
–k	Allows K&R compatibility	–pk	2-19, 4-18
–l (lowercase L)	Generates .pp file†	–pl	2-19, 2-35
–n	Suppresses #line directives	–pn	2-19
–o	Preprocesses only	–po	2-19
–q	Suppresses progress messages (quiet)	–q	2-15
–r	Generates error listing	–pr	2-20
–tf	Relaxes prototype checking	–tf	2-21
–tp	Relaxes pointer combination	–tp	2-21
–u*name*	Undefines macro *name*	–u*name*	2-15
–w	Suppresses warning messages	–pw	2-20, 2-49
–x	Enables inlining of user functions (implies –o2)	–x2	2-20, 2-41
–x0	Disables function inlining	–x0	2-20, 2-41
–?	Enables trigraph expansion	–p?	2-18

† When running ac500 standalone and using –l to generate a preprocessed listing file, you can specify the name of the file as the third filename on the command line. This filename can appear anywhere on the command line after the names of the source file and intermediate file.

B.2.1 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.

   ```
   cl500 –po file.c
   ```

   If you want to use the –d, –u, or –i options, use them on this first pass. This pass produces a preprocessed output file called file.pp. For more information about the preprocessor, see Section 2.2, *Controlling the Preprocessor*, on page 2-30.

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

   ```
   cl500 file.pp
   ```

   You can use any other options on this final pass.

B.3 Invoking the Optimizer Individually

The second step in compiling a ’C54x 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 optimized intermediate file has the same format as the original intermediate file, but it enables the code generator to produce more efficient code.

To invoke the optimizer, enter:

opt500 [*input file* [*output file*]] [*options*]

opt500 is the command that invokes the optimizer.

input file names the intermediate file produced by the parser. The optimizer assumes that the extension is *.if*. If you don’t specify an input file, the optimizer prompts you for one.

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. Table B–2 on page B-8 shows the optimizer options, the shell options, and the corresponding functions. Subsection 2.3.1, *Optimization Levels*, on page 2-37 provides a detailed list of the optimizations performed at each level.

Table B–2. Optimizer Options and Shell Options

Optimizer Option	Function	See Shell Option	See Page
–a	Assumes variables are aliased	–ma	2-22
–b	Disables the noninterruptible RPT instruction for moving structures	–mb	2-22
–g*register*†	Reserves global register	–r*register*	2-15
–h*n*	Controls assumptions about library function calls	–ol*n*	2-23
–i*nn*	Sets automatic inlining size threshold (–o3 only)	–oi*size*	2-22, 2-40
–k	Allows K&R compatibility	–pk	2-19, 4-18
–n*n*	Generates optimization information file (–o3 only)	–on*n*	2-23
–o0	Optimizes at level 0 (register optimization)	–o0	2-22
–o1	Optimizes at level 1 (level 0 plus local optimization)	–o1	2-22
–o2	Optimizes at level 2 (level 1 plus global optimization)	–o2	2-22
–o3	Optimizes at level 3 (level 2 plus file optimization)	–o3	2-22
–q	Suppresses progress messages (quiet)	–q	2-15
–s	Interlists C source	–os	2-46

† The **–g** option tells the optimizer that the register named is reserved for global use. See Section 4.5, *Creating Global Register Variables*, on page 4-7 for more information.

B.4 Invoking the Code Generator Individually

The third step in compiling a 'C54x C program is to invoke the code generator. As Figure B–1 on page B-2 shows, the code generator converts the intermediate file produced by the parser or the optimizer into an assembly language source file. You can modify this output file or use it as input for the assembler. The code generator produces reentrant relocatable code, which, after assembling and linking, can be stored in ROM.

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

cg500 [*input file* [*output file* [*tempfile*]]] [*options*]

cg500 is the command that invokes the code generator.

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

output file names the assembly language source file that the code generator creates. If you don't supply a filename for the output file, the code generator uses the input filename with the extension of *.asm*.

tempfile names a temporary file that the code generator creates and uses. If you don't supply a filename for the temporary file, the code generator uses the input filename with the extension *.tmp*. The code generator deletes this file after using it.

options affect the way the code generator processes the input file. Each option available for the standalone code generator mode has a corresponding shell option that performs the same function. The following table shows the code generator options, the shell options, and the corresponding functions.

Table B–3. Code Generator Options and Shell Options

Code Generator Options	Function	See	
		Shell Options	Page
–a	Assumes variables are aliased	–ma	2-22
–b	Disables the noninterruptible RPT instruction for moving structures	–mb	2-22
–f	All calls are far calls and all returns will be far returns	–mf	2-22
–g*register*†	Reserves global register	–r*register*	2-15
–n	Reenables optimizations disabled by symbolic debugging	–mn	2-22
–o	Enables C source level debugging	–g	2-14
–q	Suppresses progress messages (quiet)	–q	2-15
–s	Optimizes for space instead of for speed	–ms	2-22
–x	Avoids first run silicon bugs	–mx	2-22
–z‡	Retains the input file	—	—

† The **–g** option tells the code generator that the register named is reserved for global use. See Section 4.5, *Creating Global Register Variables*, on page 4-7 for more information.

‡ The **–z** option tells the code generator to retain the input file (the intermediate file created by the parser or the optimizer). If you do not specify the –z option, the intermediate file is deleted.

B.5 Invoking the Interlist Utility Individually

Note: Interlisting With the Shell Program and the Optimizer

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

The fourth step in compiling a 'C54x C program is optional. After you have compiled a program, you can run the interlist utility as a standalone program. To run the interlist utility from the command line, the syntax is:

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

clist is the command that invokes the interlist utility.

asmfile names the assembly language file produced by the compiler.

outfile names the interlisted output file. If you don't supply a filename for the outfile, the interlist utility uses the assembly language filename with the extension *.cl*.

options control the operation of the utility as follows:

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

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

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

```
cl500 –gk –mn function
```

This compiles, produces symbolic debugging directives, and keeps the assembly language file. To produce an interlist file, enter:

```
clist –r function
```

This creates an interlist file and removes the symbolic debugging directives. The output from this example is function.cl.

Appendix C

Glossary

A

ANSI (American National Standards Institute): A board that approves American National Standards.

absolute lister: A debugging tool that allows you to create assembler listings that contain absolute addresses.

aliasing: The ability for a single object to be accessed in more than one way, such as when two pointers point to a single object. It can disrupt optimization because any indirect reference could potentially refer to any other object.

alignment: A process in which the linker places an output section at an address that falls on an *n*-bit boundary, where *n* is a power of 2. You can specify alignment with the SECTIONS linker directive.

allocation: A process in which the linker calculates the final memory addresses of output sections.

archive library: A collection of individual files that have been grouped into a single file.

archiver: A software program that allows you to collect several individual files into a single file called an archive library. The archiver also allows you to delete, extract, or replace members of the archive library, as well as to add new members.

argument block: The part of the local frame used to pass arguments to other functions.

assembler: A software program that creates a machine-language program from a source file that contains assembly language instructions, directives, and macro definitions. The assembler substitutes absolute operation codes for symbolic operation codes, and absolute or relocatable addresses for symbolic addresses.

assignment statement: A statement that assigns a value to a variable.

autoinitialization: The process of initializing global C variables (contained in the .cinit section) before beginning program execution.

auxiliary entry: An extra entry that a symbol may have in the symbol table. The entry contains additional information about the symbol (whether the symbol is a filename, a section name, a function name, and so on).

B

banner: Information in a compiler listing that denotes one pass through the compiler. The information identifies the compiler.

binding: A process in which you specify a distinct address for an output section or a symbol.

block: A set of declarations and statements that are grouped together with braces.

.bss: One of the default COFF sections. You can use the .bss directive to reserve a specified amount of space in the memory map that can later be used for storing data. The .bss section is uninitialized.

byte: Traditionally, a sequence of eight adjacent bits operated upon as a unit. However, the 'C54x byte is 16 bits.

Note: 'C54x Byte Is 16 Bits

By ANSI C definition, the sizeof operator yields the number of bytes required to store an object. ANSI further stipulates that when sizeof is applied to char, the result is 1. Since the 'C54x char is 16 bits (to make it separately addressable), a byte is also 16 bits. This yields results you may not expect; for example, sizeof (int) = = 1 (not 2). 'C54x bytes and words are equivalent (16 bits).

C

C compiler: A program that translates C source statements into assembly language source statements.

code generator: A compiler tool that takes the intermediate file produced by the parser or the optimizer and produces an assembly language source file.

command file: A file that contains options, filenames, directives, or comments for the compiler or linker.

comment: A source statement (or portion of a source statement) that is used to document or improve readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.

common object file format (COFF): An object file format that promotes modular programming by supporting the concept of *sections*.

conditional processing: A method of processing one block of source code or an alternate block of source code, according to the evaluation of a specified expression.

configured memory: Memory that the linker has specified for allocation.

constant: A numeric value that can be used as an operand.

cross-reference listing: An output file created by the assembler that lists the symbols that were defined, what line they were defined on, which lines referenced them, and their final values.

D

.data: One of the default COFF sections. The .data section is an initialized section that contains initialized data. You can use the .data directive to assemble code into the .data section.

data memory: A section of memory that contains external variables, static variables, and the system stack.

dynamic memory allocation: Memory allocation created by several functions (such as malloc, calloc, and realloc) that allows you to dynamically allocate memory for variables at runtime. This is accomplished by declaring a large memory pool, or heap, and then using the functions to allocate memory from the heap.

E

emulator: A hardware development system that accepts the same inputs and produces the same outputs as the ’C54x device.

entry point: The location in target memory where the program begins execution.

executable module: An object file that has been linked and can be run in a target system.

expression: A constant, a symbol, or a series of constants and symbols separated by arithmetic operators.

external symbol: A symbol that is used in the current program module but defined in a different program module.

F

field: For the ’C54x, a software-configurable data type whose length can be programmed to be any value in the range of 1–16 bits.

file header: A portion of a COFF object file that contains general information about the object file, such as the number of section headers, the type of system the object file can be downloaded to, the number of symbols in the symbol table, and the symbol table’s starting address.

function inlining: Code for a function is inserted at the point of the call. Function inlining saves the overhead (the code necessary to enter and exit the function) of a function call. Function inlining also allows the optimizer to make the function code as efficient as possible in the context of the surrounding code.

G

global symbol: A symbol that is either defined in the current module and accessed in another, or accessed in the current module but defined in another.

GROUP: An option of the SECTIONS directive that forces specified output sections to be allocated contiguously (as a group).

H

hex conversion utility: A utility that converts COFF object files into one of several standard ASCII hexadecimal formats, suitable for loading into an EPROM programmer.

hole: An area between the input sections that compose an output section that contains no code or data.

I

incremental linking: The linking of files that have already been linked.

initialized section: A COFF section that contains executable code or initialized data. An initialized section can be built with the .data, .text, or .sect directive.

input section: A section from an object file that will be linked into an executable module.

integrated preprocessor: A preprocessor that is combined with the parser, allowing for faster compilation.

interlist utility: A utility that includes (as comments) your original C source statements with the assembly language output from the assembler.

K

K&R: Kernighan and Ritchie C, the de facto standard as defined in the second edition of *The C Programming Language*. Most K&R C programs written for earlier non-ANSI C compilers should correctly compile and run without modification.

L

label: A symbol that begins in column 1 of an assembler source statement and corresponds to the address of that statement. This is the only assembler statement that can begin in column 1.

line number entry: An item in a COFF output module that maps lines of assembly code back to the original C source file that created them.

linker: A software tool that combines object files to form an object module that can be allocated into system memory and executed by the device.

listing file: An output file created by the assembler that lists source statements, their line numbers, and their effects on the section program counter (SPC).

loader: A device that places an executable module into system memory.

M

macro: A user-defined routine that is used as an instruction.

macro call: The process of invoking a macro.

macro definition: A block of source statements that define the name and the code that make up a macro.

macro expansion: The source statements that are substituted for the macro call and are subsequently assembled.

macro library: An archive library composed of macros. Each file in the library must contain one macro; its name must be the same as the macro name it defines, and it must have an extension of .asm.

map file: An output file, created by the linker, that shows the memory configuration, section composition, and section allocation, as well as symbols and the addresses at which the symbols were defined.

memory map: A map of target system memory space, which is partitioned off into functional blocks.

N

named section: An initialized section that is defined with a .sect directive, or an uninitialized section that is defined with a .usect directive.

O

object file: A file that has been assembled or linked and contains machine-language object code.

object library: An archive library made up of individual object files.

object module: A group of object files that have been linked together to create an executable program.

optional header: A portion of a COFF object file that the linker uses to perform relocation at download time.

options: Command-line parameters that allow you to request additional or specific functions when you invoke a software tool.

optimizer: A software tool that improves the execution speed and reduces the size of C programs by rewriting pieces of code to take advantage of the ’C54x architecture.

output module: A linked, executable object file that can be downloaded and executed on a target system.

output section: A final, allocated section in a linked, executable module.

overlay pages: Multiple areas of physical memory that occupy the same address space at different times. ’C54x devices can map different pages into the same address space in response to hardware select signals.

P

parser: A software tool that reads the source file, performs preprocessor functions, checks the syntax, and produces an intermediate file that can be used as input for the optimizer or code generator.

partial linking: The linking of a file that will be linked again.

pragma directive: A pragma directive tells the preprocessor how to treat functions.

preprocessor: A software tool that expands macro definitions, included files, conditional compilation, and preprocessor directives.

program memory: A section of memory that contains executable code.

R

RAM autoinitialization model: An autoinitialization method used by the linker when linking C code. The linker uses this model when you invoke the linker with the –cr option. The RAM model allows variables to be initialized at load time instead of runtime.

relocation: A process in which the linker adjusts all the references to a symbol when the symbol's address changes.

ROM autoinitialization model: An autoinitialization method used by the linker when linking C code. The linker uses this model when you invoke the linker with the –c option. In the ROM model, the linker loads the .cinit section of data tables into memory, and variables are initialized at runtime.

runtime environment: The conditions within which the system operates. These include memory and register conventions, stack organization, function call conventions, and system initialization.

runtime-support functions: Standard ANSI functions that perform tasks that are not part of the C language, such as memory allocation, string conversion, and string searches.

runtime-support library: A library file, rts.src, that contains the source for the runtime-support functions as well as for other functions and routines.

S

section: A relocatable block of code or data that will occupy contiguous space in the memory map.

section header: A portion of a COFF object file that contains information about a section in the file. Each section has its own header; the header points to the section's starting address, contains the section's size, etc.

section program counter: See SPC.

sign extend: A process that fills the unused MSBs of a value with the value's sign bit.

source file: A file that contains C code or assembly language code that will be compiled or assembled to form an object file.

SPC (section program counter): An element of the assembler that keeps track of the current memory location within a section; each section has its own SPC.

static variable: A kind of variable whose scope is confined to a function or a program. The values of static variables are not discarded when the function or program is exited; their previous value is resumed when the function or program is reentered.

storage class: An entry in the symbol table that indicates how a symbol must be accessed.

string table: A matrix that stores symbol names that are longer than eight characters (symbol names of eight characters or longer cannot be stored in the symbol table; instead, they are stored in the string table.) The name portion of a symbol's entry points to the location of the string in the string table.

structure: A collection of one or more variables grouped together under a single name.

symbol: A string of alphanumeric characters that represents an address or a value.

symbol table: A portion of a COFF object file that contains information about the symbols that are defined and used by the file.

T

target memory: The physical memory in a 'C54x-based system into which executable object code is loaded.

.text: One of the default COFF sections; an initialized section that contains executable code. You can use the .text directive to assemble code into the .text section.

U

unconfigured memory: Memory that is not defined as part of the memory map and cannot be loaded with code or data.

uninitialized section: A COFF section that reserves space in the memory map but that has no actual contents. These sections are built up with the .bss and .usect directives.

union variable: A variable that may hold (at different times) objects of different types and sizes.

unsigned value: A kind of value that is treated as a positive number, regardless of its actual sign.

V

variable: A symbol representing a quantity that may assume any of a set of values.

W

well-defined expression: An expression that contains only symbols or assembly-time constants that have been defined before they appear in the expression.

word: A sequence of bits or characters that is stored, addressed, transmitted, and operated on as a unit. A 16-bit addressable location in target memory.

Index

Sy

A

B

C

D

E

F

G

H

I

J

K

L

P

Q

R

S

T

U

V

W

X

Z