TEXAS INSTRUMENTS

TMS370 and TMS370C8 8-Bit Microcontroller Family Optimizing C Compiler

User's Guide

1996 Microprocessor Development Systems

Texas Instruments

Printed in U.S.A., April 1996
2547306-9761 revision C*

SPNU022C

User's Guide

TMS370 and TMS370C8 8-Bit Microcontroller Family Optimizing C Compiler

1996

TMS370 and TMS370C8 8-Bit Microcontroller Family Optimizing C Compiler User's Guide

SPNU022C
April 1996

TEXAS INSTRUMENTS

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.

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.

Contents

Figures

Tables

Examples

Notes, Cautions, and Warnings

Preface

Read This First

About This Manual

The *TMS370 and TMS370C8 8-Bit Microcontroller Family Optimizing C Compiler User's Guide* tells you how to use these compiler tools:

- Parser
- Optimizer
- Code generator
- Library-build utility

The TMS370/C8 8-bit C compiler accepts American National Standards Institute (ANSI) standard C source code and produces assembly language source code for the TMS370/C8 8-bit devices. This user's guide discusses the characteristics of the C compiler. It assumes that you already know how to write C programs. *The C Programming Language* (second edition), by Brian W. Kernighan and Dennis M. Ritchie, describes C based on the ANSI C standard. You can use the Kernighan and Ritchie (hereafter referred to as K&R) book as a supplement to this manual.

Before you use the information about the C compiler in this user's guide, you should read the *TMS370 and TMS370C8 8-Bit Microcontroller Family Code Generation Tools Getting Started Guide* to install the C compiler tools.

Notational Conventions

This document uses the following conventions:

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

  Here is a sample 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 face** typeface and parameters are in *italics*. Portions of a syntax that are in bold face must be entered as shown; portions of a syntax that are in italics describe the type of information that should be entered. Syntax that will be entered on a command line is centered in a bounded box:

  cl370 [*options*] [*filenames*] [*–z* [*link_options*] [*object files*]]

Syntax used in a text file is left justified in an unbounded box:

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

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

  hex370 [*options*] *filename*

The hex370 command has two parameters. The second parameter, *filename*, is required. The first parameter, *–options*, is optional.

- Braces ({ and }) indicate that you must choose one of the parameters within the braces; you don't enter the braces themselves. (In the *inline* example above in the unbounded box, you would enter the braces because they are in a bold face typeface.) This is an example of a command with braces that are not included in the actual syntax but indicate that you must specify either the –c or –cr option:

  lnk370 {**–c** | **–cr**} *filenames* [**–o** *name.out*] **–l** *libraryname*

- The TMS370 and TMS370C8 8-bit microcontroller devices are collectively referred to as '370/C8 or TMS370/C8.

Related Documentation

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

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

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

You may find this document helpful as well:

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

Related Documentation From Texas Instruments

The following books, which describe the TMS370, TMS370C8, and related support tools, are available from Texas Instruments. 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 (located on the bottom-right corner of the back cover):

TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide (literature number SPNU010) describes the assembly language tools (assembler, linker, and other tools used to develop assembly code), assembler directives, macros, common object file format, and symbolic debugging directives for the TMS370/C8 8-bit family of devices.

TMS370 Family C Source Debugger User's Guide (literature number SPNU028) tells you how to invoke the '370 XDS/22 emulator and application board versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints, and includes a tutorial that introduces basic debugger functionality. It also includes an advanced tutorial that introduces the breakpoint, trace, and timing features.

TMS370C8 C Source Debugger User's Guide (literature number SPNU063) tells you how to invoke the TMS370C8 XDS emulator, compact development tool (CDT), 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.

TMS370 Microcontroller Family User's Guide (literature number SPNU127) discusses hardware aspects of the TMS370 family members, such as pin functions, architecture, module options, stack operation, and interfaces. The manual also contains the TMS370 assembly language instruction set.

TMS370C8 Central Processing Unit, System, and Instruction Set Reference Guide (literature number SPNU042) discusses the TMS370C8 architecture, features, operation, and assembly language instruction set; it also includes helpful information about implementing a TMS370C8-based microcontroller design.

If You Need Assistance. . .

If you want to. . .	Do this. . .
Request more information about Texas Instruments microcontroller products	Write to: Texas Instruments Incorporated Market Communications Manager, MS 6101 P.O. Box 1443 Houston, Texas 77251–1443
Order Texas Instruments documentation	Call TI Literature Response Center: **(800) 477–8924**
Ask questions about product operation or report suspected problems	Call the Microcontroller Hotline: **(713) 274–2370**
Report mistakes or make comments about this or any other TI documentation	Send your comments to: **comments@books.sc.ti.com**
Please mention the full title of the book and the date of publication (from the spine and/or front cover) in your correspondence.	Texas Instruments Incorporated Technical Publications Mgr., MS 702 P.O. Box 1443 Houston, Texas 77251–1443
Visit TI online, including TI&ME™, your own customized web page.	**http://www.ti.com**

Trademarks

Chapter 1

Introduction

The TMS370/C8 8-bit microcontroller family includes two generations of CMOS 8-bit microcontrollers. The microcontrollers are listed below by generation:

First generation:

TMS370Cx0x	TMS370Cx5x	TMS370Cx9x
TMS370Cx1x	TMS370Cx6x	TMS370CxAx
TMS370Cx2x	TMS370Cx7x	TMS370CxBx
TMS370Cx3x	TMS370Cx8x	TMS370CxCx
TMS370Cx4x		

Second generation: TMS370C8

Both generations are supported by a single set of code generation tools, including an optimizing C compiler, an assembler, a linker, and assorted utilities. This provides upward code compatibility between the first and second generations of 8-bit devices. The two generations use separate debugging tools.

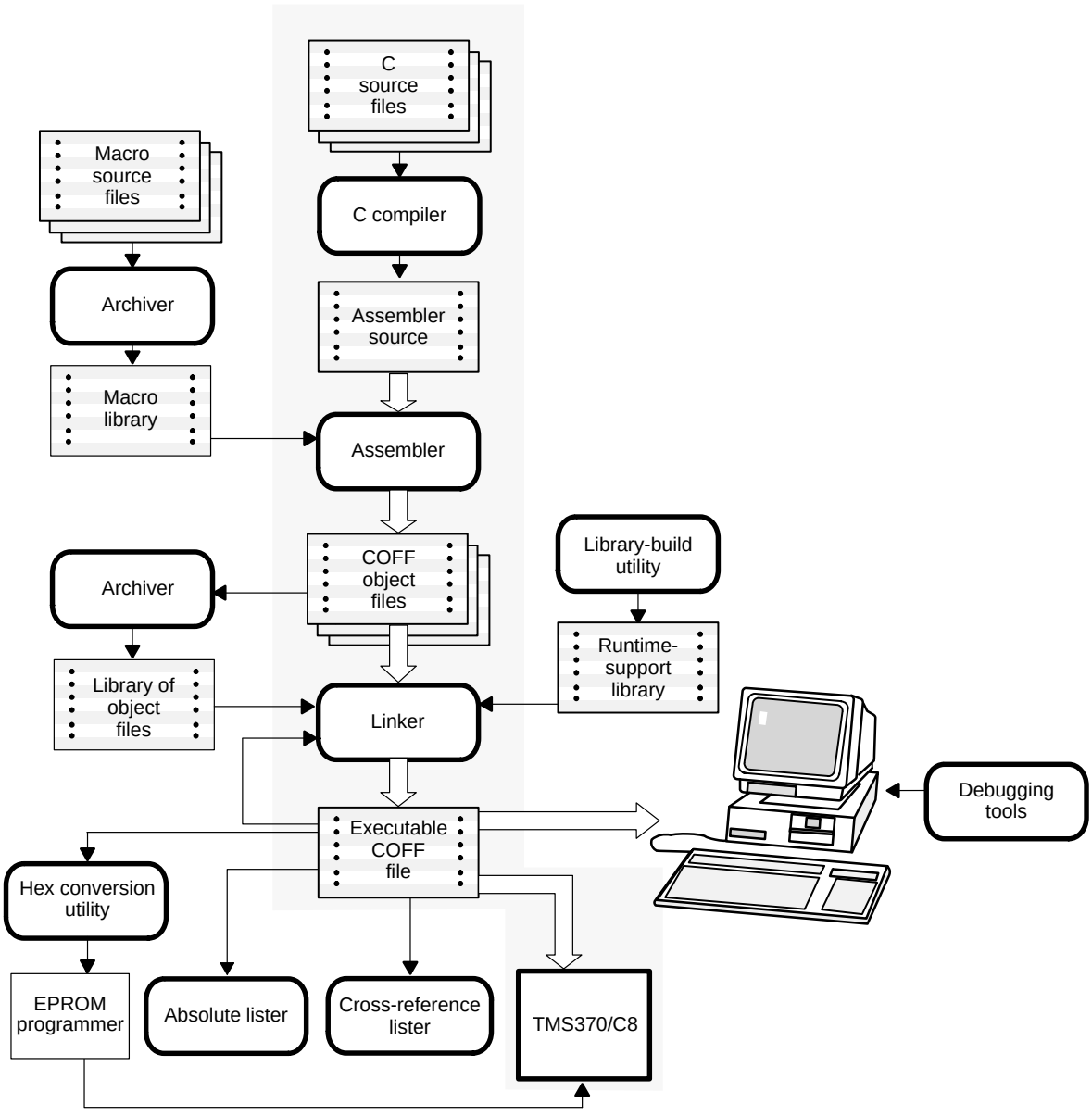
The TMS370C8 family has an enhanced instruction set for greater functionality and faster execution. This chapter provides an overview of these tools and introduces the features of the optimizing C compiler.

Topic **Page**

1.1 Software Development Tools Overview

Figure 1–1 illustrates the ’370/C8 software development flow. The shaded portion of the figure highlights the most common path of software development for C language programs; the other portions are optional.

Figure 1–1. TMS370/C8 Software Development Flow



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

- The **C compiler** accepts C source code and produces ’370/C8 assembly language source code. A **shell program**, an **optimizer**, and an **interlist utility** are parts of the compiler:
  - The shell program enables you to compile, assemble, and link source modules in one step.
  - The optimizer modifies code to improve the efficiency of C programs.
  - The interlist utility interlists C source statements with assembly language output.

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

- The **assembler** translates assembly language source files into machine language object files. The machine language is based on common object file format (COFF). The *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User’s Guide* explains how to use the assembler.

- The **linker** combines object files into a single executable object module. As it creates the executable module, it performs relocation and resolves external references. The linker accepts relocatable COFF object files and object libraries as input. See Chapter 3, *Linking C Code*, for information about invoking the linker. See the *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User’s Guide* for a complete description of the linker.

- The **archiver** allows you to collect a group of files into a single archive file, called a *library.* Additionally, the archiver allows you to modify a library by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is building a library of object modules. rts.lib is the runtime-support object library shipped with the C compiler. The *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User’s Guide* explains how to use the archiver.

- You can use the **library-build utility** to build your own customized runtime-support library. See Chapter 7, *Library-Build Utility,* for more information. Standard runtime-support library functions are provided as source code located in rts.src.

  The rts.lib and the rts_c8.lib **runtime support libraries** contain ANSI standard runtime-support functions, compiler-utility functions, and floating-point arithmetic functions that can be used for the ’370 and ’370C8, respectively.

- The ’370/C8 debugger accepts COFF files as input, but some EPROM programmers do not. The **hex conversion utility** converts a COFF object file into TI-Tagged, ASCII-hex, Intel, Motorola-S, or Tektronix object format. The converted file can be downloaded to an EPROM programmer. The *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User’s Guide* explains how to use the hex conversion utility.

- The **absolute lister** is a debugging tool. It accepts linked object files as input and creates .abs files as output. You can put these .abs files together to produce a listing that contains absolute rather than relative addresses. Without the absolute lister, producing such a listing would be tedious and require many manual operations. The *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User’s Guide* explains how to use the absolute lister.

- The **cross-reference utility** uses object files to produce a cross-reference listing showing symbols, their definitions, and their references in the linked source files. The *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User’s Guide* explains how to use the cross-reference utility.

- The main product of this development process is a module that can be executed on a **TMS370/C8** device. You can use one of several debugging tools to refine and correct your code. Available products include:

  - An instruction-accurate and clock-accurate software simulator
  - An emulator
  - A compact development tool
  - An application board (TMS370 family only)

  For information about these debugging tools, see the C source debugger user’s guide.

1.2 C Compiler Overview

The ’370/C8 C compiler is a full-featured optimizing compiler that translates standard ANSI C programs into ’370/C8 assembly language source. The following list describes key characteristics of the compiler:

- **ANSI-standard C**

  The ’370/C8 compiler fully conforms to the ANSI C standard as defined by the ANSI specification and described in the second edition of Kernighan and Ritchie’s *The C Programming Language* (K&R). The ANSI C standard includes extensions to C that provide maximum portability and increased capability.

- **ANSI-standard runtime support**

  The compiler tools come with a complete runtime library. All library functions conform to the ANSI C library standard. The library includes functions for string manipulation, dynamic memory allocation, data conversion, timekeeping, trigonometry, and 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.*

- **Assembly source output**

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

- **Automatic overlay capability**

  The compiler, assembler, and the linker automatically overlay register blocks and uninitialized memory sections that contain automatic data. The compiler generates information into the COFF file that the linker uses to construct a call graph. From the call graph, the linker can determine which functions are never active at the same time. It then allocates the automatic data space for these functions into the same location.

- **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 rich support for source-level debugging.

- **Compiler shell program**

  The compiler tools include a shell program, which enables you to compile, assemble, and link programs in a single step. For more information, refer to Section 2.1, *Using the Shell Program to Compile, Assemble*, *and Link,* 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*.

- **Integrated preprocessor**

  The C preprocessor is integrated with the parser, allowing for faster compilation. Standalone preprocessing or preprocessed listing is also available. For more information, see Section 2.3, *Controlling the Preprocessor*, on page 2-19.

- **Optimization**

  The compiler uses a sophisticated optimization pass that employs several advanced techniques for generating efficient, compact code from C source. General optimizations can be applied to any C code, and ’370/C8-specific optimizations take advantage of the features specific to the ’370/C8 architecture. For more information about the C compiler’s optimization techniques, see Section 2.4, *Using the C Compiler Optimizer,* on page 2-24 and Appendix A, *Optimizations*.

- **ROM-able code**

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

- **Source interlist utility**

  The compiler tools include 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 2.6, *Using the Interlist Utility,* on page 2-40.

Chapter 2

C Compiler Description

Translating your source program into code that the ’370/C8 can execute is a multistep process. You must compile, assemble, and link your source files to create an executable object file. The ’370/C8 compiler tools contain a special shell program, cl370, that enables you to execute all of these steps with one command. This chapter provides a complete description of how to use the shell program to compile, assemble, and link your programs.

This chapter also describes the shell’s preprocessor, the optimizer, the inline function expansion features, and the interlist utility:

- During compilation, your code is run through the preprocessor, which is part of the parser. The shell program allows you to control the preprocessor with macros and various other preprocessor directives.
- The C compiler includes an optimizer that can be optionally invoked to allow you to produce highly optimized code.
- Inline function expansion allows you to save the overhead on a function call and enable further optimizations.
- The compiler tools include a utility that interlists your original C source statements into the compiler’s assembly language output. This enables you to inspect the assembly language code generated for each C statement.

Topic **Page**

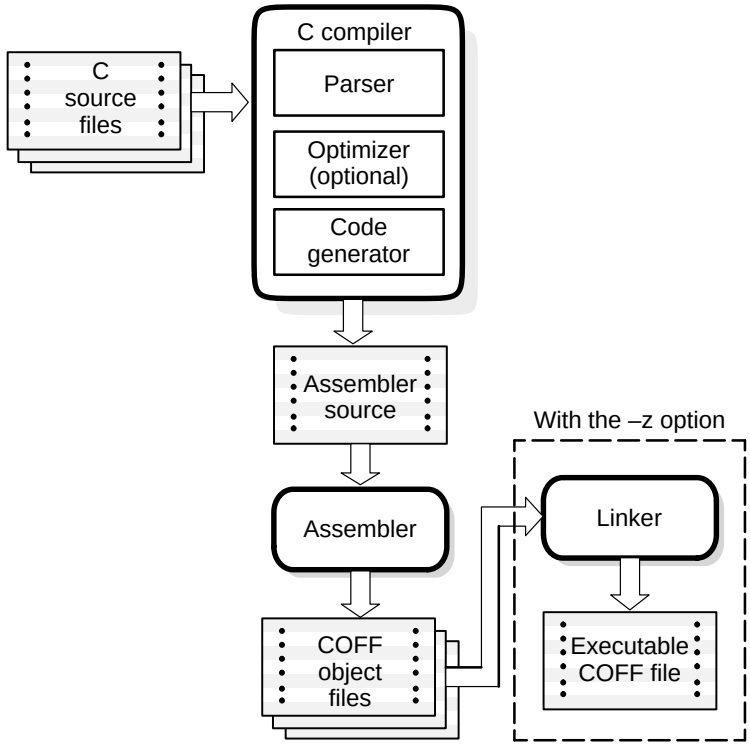
2.1 Using the Shell Program to Compile, Assemble, and Link

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

- The **compiler**, which includes the parser, optimizer, and code generator, accepts C source code and produces ’370/C8 assembly language source code.
- The **assembler** generates a COFF object file.
- The **linker** links your files to create an executable object file. The linker is optional at this point. You can compile and assemble various files with the shell and link them later. See Chapter 3, *Linking C Code,* for information about linking the files in a separate step.

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

Figure 2–1. The Shell Program Overview



For a complete description of the assembler and the linker, see the *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide*. For information about invoking the compiler tools individually, see Appendix B of this user's guide.

2.1.1 Invoking the C Compiler Shell

To invoke the compiler shell, enter:

cl370 [*options*] *filenames* [**–z** [*link options*] [*object files*]]

cl370	is the command that runs the compiler and the assembler.
options	affect the way the shell processes input files.
filenames	are one or more C source files, assembly source files, or object files.
–z	is the option that invokes the linker. See Chapter 3, *Linking C Code,* for more information about invoking the linker.
link options	control the linking process.
object files	name the additional object files for the linking process.

The –z option and its associated information (link options and object files) must follow all filenames and compiler options on the command line. All other shell options and filenames can be specified in any order on the command line. For example, if you wanted to compile two files named symtab and file, assemble a third file named seek.asm, and suppress progress messages (–q), you would enter:

```
cl370 –q symtab file seek.asm
```

As cl370 encounters each source file, it prints the filename in square brackets ([ ]) for C files or angle brackets (< >) for assembly language files. This example uses the –q option to suppress the additional progress information that cl370 produces. Entering the command above produces these 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 processed. The example below shows the output from compiling a single file (symtab) *without* the –q option:

```
% cl370 symtab
[symtab]
TMS370 ANSI C Compiler     Version 6.xx
Copyright (c) 1996         Texas Instruments Incorporated
   ″symtab.c″      ==>     symtab
TMS370 C Codegen           Version 6.xx
Copyright (c) 1996         Texas Instruments Incorporated
   ″symtab.c″:     ==>     symtab
TMS370 Macro Assembler     Version 6.xx
Copyright (c) 1996         Texas Instruments Incorporated
 PASS 1
 PASS 2
```

2.1.2 Specifying Filenames

The input files that you specify 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 is assumed)	C source
.asm, .abs, or .s* (extension begins with s)	assembly source
.obj	object

Files without extensions are assumed to be C source files. The conventions for filename extensions allow you to compile C files and assemble assembly files with a single command, as shown in the first example on page 2-3.

For information about how you can alter the way that the shell interprets file extensions, see subsection 2.1.3. For information about how you can alter the way that the shell names the extensions of the files that it creates, see subsection 2.1.4.

You can use wildcard characters to compile or assemble multiple files. You cannot use wildcards if you invoke the tools individually. Wildcard specifications vary by system; use the appropriate form listed in your operating system manual. For example, to compile all of the C files in a directory, enter the following:

```
cl370 *.c
```

2.1.3 Changing How the Shell Program Interprets and Names Extensions (–fa, –fc, –fo, –ea, and –eo Options)

You can use command-line options to change how the shell interprets your file extensions. If your file naming conventions do not conform to those of the shell, you can use the –fa, –fc, and –fo options to specify which files are assembly language source files, C source files, and object files. You can insert an optional space between the option and the filename. Select the appropriate option for the type of file you want to specify:

–fa*filename* for an assembly language source file

–fc*filename* for a C source file

–fo*filename* for an object file

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

```
cl370 –fc file.s –fa assy
```

You cannot use the –fa, –fc, and –fo options with wildcard specifications.

You can also use options –ea and –eo to change how the shell program interprets filename extensions and names the extensions of the files that it creates. Select the appropriate option for the type of extension you want to specify:

–ea[**.**] *new extension* for an assembly language file

–eo[**.**] *new extension* for an object file

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

```
cl370 –ea .rrr –eo .o fit.rrr
```

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

```
cl370 –earrr –eoo fit.rrr
```

You can use these options to designate parts of your code that need to be compiled differently. The –ea and –eo options must precede any filenames on the command line. You can use wildcard specifications with these options.

2.1.4 Specifying Directories (–fr, –fs, and –ft Options)

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

–fr allows you to specify a directory for object files. If you do not specify the –fr option, the shell places object files in the current directory. To specify an object file directory, type the directory's pathname on the command line after the –fr option:

```
cl370 –fr d:\object ...
```

–fs allows you to specify a directory for assembly files. If you do not specify the –fs option, the shell places assembly files in the current directory. To specify an assembly file directory, type the directory's pathname on the command line after the –fs option:

```
cl370 –fs d:\assembly ...
```

–ft allows you to specify a directory for temporary intermediate files. The –ft option overrides the TMP environment variable. (For information about the TMP environment variable, see *The TMP environment variable* discussion in the *TMS370 and TMS370C8 8-Bit Microcontroller Family Code Generation Tools Getting Started Guide*). To specify a temporary directory, type the directory's pathname on the command line after the –ft option:

```
cl370 –ft c:\temp ...
```

2.2 Changing the Compiler's Behavior With Options

Command-line 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, the type-checking options, and the assembler options.

Options are either single letters or two-letter pairs, are *not* case sensitive, and 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, –mi and –mc can be combined as –mic. Options that have parameters, such as –u*name* and –i*dir*, cannot be combined and must be specified separately.

You can set up default options for the shell by using the C_OPTION environment variable. For a detailed description of the C_OPTION environment variable, see the *TMS370 and TMS370C8 8-Bit Microcontroller Family Code Generation Tools Getting Started Guide*.

Table 2–1 summarizes all shell and linker options. Use the page references in the table to refer to more complete descriptions of the options.

For an online summary of all the shell and linker options, enter **cl370** with no parameters on the command line.

Table 2–1. Shell Options Summary

(a) Options that control the compiler shell

Option	Effect	Page(s)
–@*filename*	interpret contents of a file as an extension to the command line	2-14
–c	disable linking (negate –z)	2-14, 3-4
–c8	generate TMS370C8 code	2-14
–d*name*[=*def*]	predefine *name*	2-14
–g	enable symbolic debugging	2-14
–i*directory*	define #include search path	2-14, 2-21
–k	keep the assembly language (.asm) file	2-14
–n	compile only	2-15
–q	suppress progress messages (quiet)	2-15
–qq	suppress all messages (super quiet)	2-15
–s	interlist C source statements with compiler-generated assembly language	2-40
–ss	interlist optimizer comments with C source and assembly statements	2-16, 2-41
–u*name*	undefine *name*	2-16
–z	enable linking	2-16

(b) Options that relax ANSI C type-checking

Option	Effect	Page
–tf	relax prototype checking	2-17
–tp	relax pointer combination checking	2-17

Table 2–1. Shell Options Summary (Continued)

(c) Options that control the parser

Option	Effect	Page
–pe	treat code-E errors as warnings	2-43
–pf	generate function prototype listing file	2-23
–pk	allow K&R compatibility	4-21
–pl	generate preprocessed listing (.pp file)	2-22
–pm	combine source files to perform program-level optimization	2-26
–pn	suppress #line directives in .pp file	2-22
–po	preprocess only	2-22
–pw0	disable all warning messages	2-43
–pw1	enable serious warning messages (default)	2-43
–pw2	enable all warning messages	2-43
–p?	enable trigraph expansion	2-23

(d) Options that affect the C runtime model

Option	Effect	Page
–ma	assume variables are aliased	2-15
–mc	disable generation of automatic overlay variables	2-14
–mi	disable generation of .icalls directive when an indirect call is encountered	2-15
–mn	reenable optimizations disabled by the –g option	2-15, 2-32
–mu	prevent optimizing unsigned 8-bit loop-induction variables	2-15

Table 2–1. Shell Options Summary (Continued)

(e) Options that control the level of optimization

Option	Effect	Page
–o0	optimize registers	2-24
–o1	use –o0 optimizations *and* optimize locally	2-24
–o2 or –o	use –o1 optimizations *and* optimize globally	2-25
–o3	use –o2 optimizations *and* optimize file	2-25
–ois*ize*	set automatic inlining size (–o3 only)	2-35

(f) Options that control the library functions

Option	Effect	Page
–ol0 (–oL0)	inform the optimizer that your file alters a standard library function	2-31
–ol1 (–oL1)	inform the optimizer that your file defines a standard library function	2-31
–ol2 (–oL2)	inform the optimizer that your file does not define or alter library functions	2-31

(g) Options that specify the optimizer information levels

Option	Effect	Page
–on0	disable optimizer information file	2-31
–on1	produce optimizer information file	2-31
–on2	produce verbose optimizer information file	2-31

Table 2–1. Shell Options Summary (Continued)

(h) Options that control the aggressiveness of program-level optimization

Option	Effect	Page
–op0	This module uses functions and variables that are called or modified from outside the source code provided to the compiler.	2-29
–op1	This module uses variables modified from outside the source code provided to the compiler but does not use functions called from outside the source code.	2-29
–op2	This module uses no functions or variables that are called or modified from outside the source code provided to the compiler (default)	2-29
–op3	This module uses functions that are called from outside the source code provided to the compiler but does not use variables modified from outside the source code.	2-29

(i) Options that control the definition-controlled inline function expansion

Option	Effect	Page
–x0	disable inline expansion of intrinsic operators	2-35
–x1	enable inline expansion of intrinsic operators (default)	2-35
–x2 or –x	define the symbol _INLINE and invoke the optimizer with –o2	2-35

(j) Options that control the assembler

Option	Effect	Page
–aa	enable absolute listing	2-18
–al	generate an assembly listing file	2-18
–as	put labels in symbol table	2-18
–ax	generate the cross-reference file	2-18

Table 2–1. Shell Options Summary (Continued)

(k) Options that change the default file extensions

Option	Effect	Page
–ea*extension*	set default extension for assembly files	2-5
–eo*extension*	set default extension for object files	2-5

(l) Options that specify file and directory names

Option	Effect	Page
–fa*filename*	identify assembly language file (default for .asm or .s*)	2-5
–fc*filename*	identify C source file (default for .c or no extension)	2-5
–fo*filename*	identify object file (default for .o*)	2-5
–fr*directory*	specify object file directory	2-6
–fs*directory*	specify assembly file directory	2-6
–ft*directory*	specify temporary file directory	2-6

Table 2–1. Options Summary (Continued)

(m) Options that control the linker

Options	Effect	Page
–a	generate absolute output	3-5
–ar	generate relocatable output	3-5
–c	use ROM initialization	3-5
–cr	use RAM initialization	3-5
–e *global_symbol*	define entry point	3-5
–f *fill_value*	define fill value	3-5
–g *global_symbol*	keep a *global_symbol* global (overrides –h)	3-5
–h	make global symbols static	3-5
–hstack *size*	set hardware stack size (bytes)	3-5
–heap *size*	set heap size (bytes)	3-5
–i *directory*	define library search path	3-5
–j	disable conditional linking	3-5
–l *filename*	supply library name	3-5
–m *filename*	name the map file	3-5
–n	ignore all fill specifications in memory directives	3-6
–novy	avoid processing of overlay sections and disable automatic overlay of register and memory blocks.	3-6
–o *filename*	name the output file	3-6
–q	suppress progress messages (quiet)	3-6
–r	generate relocatable output	3-6
–s	strip symbol table	3-6
–sstack *size*	set software stack size (bytes)	3-6
–u *symbol*	undefine symbol	3-6
–w	display a message when an undefined output section is created	3-6
–x	force rereading of libraries	3-6

2.2.1 Frequently Used Command-Line Options

Following are detailed descriptions of command-line options that you will probably use most frequently:

–@*filename* interprets a file as an extension to the command line. This option allows you to avoid limitations on command line length imposed by the host operating system. Comments are permitted on lines that begin with #.

–c suppresses the linking option; it causes the shell not to run the linker even if –z is specified. This option is especially useful when you have –z specified in the C_OPTION environment variable and you don't want to link. For more information about the –c option, refer to page 3-4.

–c8 enables generation of TMS370C8 code. Refer to the *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide* for a description of additional TMS370C8-specific instructions for generating TMS370C8 code.

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

–g causes the compiler to generate symbolic debugging directives that are used by the C source-level debuggers and enables assembly source debugging in the assembler.

–i*directory* adds *directory* to the list of directories that the compiler searches for #include files. You can use this option a maximum of 32 times to define several directories; be sure to separate –i options with spaces. If you don't specify a directory name, the preprocessor ignores the –i option. For more information, refer to subsection 2.3.2.1, *Changing the #include File Search Path With the –i Shell Option*, page 2-21.

–k keeps the assembly language file. Normally, the shell deletes the output assembly language file after assembling completes, but using –k allows you to retain the assembly language output from the compiler.

–ma assumes that variables are aliased. The compiler assumes that pointers may alias (point to) named variables and therefore disables register optimizations when an assignment is made through a pointer when the compiler determines that there may be another pointer pointing to the same object.

–mc disables the default generation of automatic overlay directives.

–mi disables the generation of the .icalls directive by the compiler when an indirect call is encountered. When you use the –mi option, you must use the CALLS pragma to specify what functions are being called indirectly. If you do not use –mi, a function containing an indirect call is not overlayed with any other function in the program, because it is assumed that that function can call any other function.

–mn reenables the optimizations disabled by the –g option. If you use the –g option to generate symbolic debugging information, many code generator optimizations are disabled because they disrupt the debugger. Therefore, if you use the –mn option, portions of the debugger's functionality will be unreliable.

–mu prevents the optimizer from optimizing unsigned 8-bit loop-induction variables. By definition, ANSI allows unsigned variables to roll over from their maximum value back to zero. The optimizer, by default, assumes this behavior is *not* intended for 8-bit array indices within for loops and allows certain optimizations to be performed. The –mu option disables these optimizations and thus allows the rollover behavior.

–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. The output of –n is assembly language output from the compiler.

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

–qq suppresses all output except error messages.

–s invokes the interlist utility, which interlists C source statements into the assembly language output of the compiler, and allows you to inspect the code generated for each C statement. This option implies the –k option. For more information, refer to Section 2.6, *Using the Interlist Utility,* on page 2-40.

–ss interlists optimizer comments with original and compiler-generated assembly language. Please note that this option may reorganize your code substantially.

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

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

2.2.2 Options That Relax ANSI C Type-Checking

Following are options that you can use to relax some of the strict ANSI C type-checking on your code:

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

```
int func( )
```

and then later declared with a prototype, such as:

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

this generates an error because the parameter types in the prototype disagree with the default argument 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 differently qualified types 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.

2.2.3 Options That Control the Assembler

Following are assembler options that you can use with the shell:

–aa invokes the assembler with the –a assembler option, which creates an absolute listing. An absolute listing shows the absolute addresses of the object code.

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

–as invokes the assembler with the –s assembler option to put labels in the symbol table. Label definitions are written to the COFF symbol table for use with symbolic debugging.

–ax invokes the assembler with the –x assembler option to produce a symbolic cross-reference in the listing file.

For more information about assembler options, see the *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide.*

2.3 Controlling the Preprocessor

This section describes specific features of the ’370/C8 preprocessor. A general description of C preprocessing is in Section A12 of K&R. The ’370/C8 C compiler includes standard C preprocessing functions, which are built into the first pass of the compiler. The preprocessor handles:

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

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

2.3.1 Predefined Macro Names

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

Table 2–2. Predefined Macro Names

Macro Name	Description
_TMS370	Set to 1 unless the –c8 command line option is defined. If the –c8 option is defined, then _TMS370 is not defined.
_TMS370C8	Set to 1 when the –c8 command line option is defined; otherwise, _TMS370C8 is undefined.
_ _LINE_ _†	Expands to the current line number
_ _FILE_ _†	Expands to the current source filename
_ _DATE_ _†	Expands to the compilation date in the form *mm dd yyyy*
_ _TIME_ _†	Expands to the compilation time in the form *hh:mm:ss*
_INLINE	Expands to 1 under the –x or –x2 optimizer option; undefined otherwise
_ _V600_ _	Set to 1 if the currently executing version of the compiler is version 6.00 or higher
_ _STDC_ _	Set to 1 to indicate that compiler conforms to ANSI C Standard. See subsection 4.1.6, page 4-4, for exceptions to ANSI C conformance.

† Specified by the ANSI standard

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

```
printf ( ”%s %s” , _TIME_ , _DATE_);
```

translates to a line such as:

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

2.3.2 The Search Path for #include Files

The #include preprocessor directive tells the compiler to read source statements from another file. You can use the #include directive without specifying any directory information for the file. See subsection 2.3.2.1 for information on using the –i option. When specifying the file, you can enclose the filename in double quotes or in angle brackets. The filename can be a complete pathname, partial path information, or a filename with no path information.

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

1) The directory that contains the current source file. (The current source file refers to the file that is being compiled when the compiler encounters the #include directive.)

2) Directories named with the –i shell option

3) Directories set with the C_DIR environment variable

If you enclose the filename in angle brackets (< >), the compiler searches for the file in the following directories in this order:

1) Directories named with the –i shell option
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. For information on how to use the C_DIR environment variable, see the *TMS370 and TMS370C8 8-Bit Microcontroller Family Code Generation Tools Getting Started Guide.*

2.3.2.1 Changing the #include File Search Path With the –i Shell Option

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

cl370 –i *directory1* [**–i** *directory2* ...]

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

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

or

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

Assume that the complete pathname for alt.h is:

MS-DOS or OS/2 c:\370tools\files\alt.h

UNIX /370tools/files/alt.h

The table below lists the complete pathname for alt.h and shows how to invoke the compiler. Select the command for your operating system:

Operating System	Enter
MS-DOS or OS/2	`cl370 –ic:\370tools\files source.c`
UNIX	`cl370 –i/370tools/files source.c`

2.3.3 Generating a Preprocessed Listing File (–pl Option)

The –pl (lowercase L) shell option allows you to generate a preprocessed version of your source file with a *.pp* extension. The compiler's preprocessing functions perform the following operations on the source file:

- Each source line ending in a backslash (\) is joined with the following line.
- Trigraph sequences are expanded (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 expanded.

These operations correspond to translation phases 1–3 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. You can use the –pn option to suppress #line directives (see subsection 2.3.3.2).

2.3.3.1 Generating a Preprocessed Listing File Without Code Generation (–po Option)

If you use the –po shell 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. The resulting preprocessed listing file is a valid C source file that can be rerun through the compiler.

2.3.3.2 Removing the #line Directives From the Preprocessed Listing File (–pn Option)

The –pn shell option suppresses line and file information in the preprocessed listing file. The –pn option causes the #line directives of the form:

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

to be suppressed in the .pp file generated with the –po or –pl shell options. The –pn option is useful when compiling machine-generated source.

2.3.4 Creating Custom Error Messages With the #warn and #error Directives

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:

`#error` *token-sequence*

For more information, refer to Section A12.7 of K&R.

2.3.5 Enabling Trigraph Expansion (–p? Option)

Trigraphs are special escape sequences of the following form:

`??c`

The letter c represents 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. If you want to enable trigraphs, use the –p? option. For more information about trigraphs, see the ANSI specification, § 2.2.1.1, or K&R A12.1.

2.3.6 Creating a Function Prototype Listing File (–pf Option)

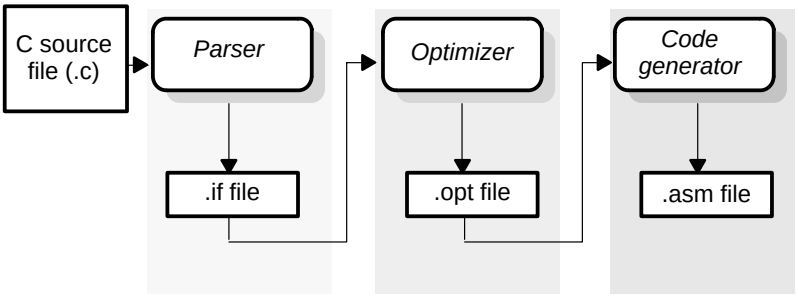
If you want to generate a listing of the procedures defined in your code, use the –pf option. When you use the –pf option, the parser creates a file containing the prototype of every procedure in all corresponding C files. Each function prototype file is named as its corresponding C file with a .pro extension.

2.4 Using the C Compiler Optimizer

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

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



The easiest way to invoke the optimizer is to use the cl370 shell program, specifying the –o*n* option on the cl370 command line. The *n* denotes the level of optimization; there are four levels: 0, 1, 2, and 3. These levels control the type and degree of optimization:

- **–o0**
  - Performs control-flow-graph simplification
  - Allocates variables to registers
  - Performs loop rotation
  - Eliminates dead code
  - Simplifies expressions and statements
  - Expands calls to functions declared inline
- **–o1**

  Performs all –o0 optimizations, plus:
  - Performs local copy/constant propagation
  - Removes dead assignments
  - Eliminates local common expressions

❑ **–o2**

Performs all –o1 optimizations, plus:

- Performs loop optimizations
- Eliminates global common subexpressions
- Eliminates global dead assignments
- Converts array references in loops to incremented pointer form
- Performs loop unrolling

The optimizer uses –o2 as the default if you compile with the –o option without specifying an optimization level.

❑ **–o3**

Performs all –o2 optimizations, plus:

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

If you use –o3, refer to subsection 2.4.1 on page 2-26 for more information.

The levels of optimization described above are performed by the standalone optimization pass. The code generator performs several additional optimizations, particularly '370/C8-specific optimizations; it does so regardless of whether or not you invoke the optimizer. These optimizations are always enabled and are not affected by the optimization level you choose.

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

You can also invoke the optimizer outside cl370; see Section B.3, *Invoking the Optimizer*, on page B-6 for information about invoking the optimizer as a separate step.

2.4.1 Using the –o3 Option

When used alone, the –o3 option performs file-level optimization. If your code contains certain kinds of files, you must combine the –o3 option with other options to effectively optimize your code. If you use the following options with –o3, you can optimize your code more specifically:

- If you want to optimize your code more fully, use the –pm option to combine source files so that the optimizer can perform program-level optimization. See subsection 2.4.1.1.
- If you have files that redefine standard library functions, use the –ol option to control file-level optimization. See subsection 2.4.1.3.
- If you want to create an optimization information file, use the –on option. See subsection 2.4.1.4.

2.4.1.1 Performing Program-Level Optimization (–pm Option)

When you use the –o3 option, you perform file-level optimization only. When you use the –o3 option with the –pm option, you perform program-level optimization.

With program-level optimization, the compiler combines all of your C source files into one intermediate file (called a module), which is then passed to the optimization and code generation passes of the compiler. Because the compiler can see the entire C program, it performs several optimizations that are rarely applied during file-level optimization:

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

If you have any assembly functions in your program, you need to exercise caution when using the –pm option. The compiler sees only the C source code and not any hand-coded assembly code that may be present. Because the compiler cannot see the assembly code's calls and variable modifications to C functions, the –pm option optimizes out those C functions. To keep these functions from being optimized out by the compiler, place the FUNC_EXT_CALLED pragma before any declaration or reference to the function that you want to keep:

```
#pragma FUNC_EXT_CALLED (func)
```

The func argument is the name of the C function that is called by hand-coded assembly language, a library, or some other unsupplied source. For more information about FUNC_EXT_CALLED, see subsection 4.6.9, page 4-18.

Another approach you can take when you use assembly functions in your program is to use the –op option with the –pm and –o3 options. The –op option (described in subsection 2.4.1.2) controls the aggressiveness of program-level optimization.

If you combine the FUNC_EXT_CALLED pragma and options incorrectly, the compiler may not recognize that you have calls and modifications from assembly code to C functions and variables. Since the compiler cannot see the assembly code's calls and variable modifications to C functions, the –pm option will optimize out those C functions. If any of these situations apply to your application, use the suggested solution:

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

Solution Compile with –pm –o3 –op2 to tell the compiler that outside functions do not call C functions or modify C variables. See subsection 2.4.1.2 for information about the –op2 option.

If you compile with the –pm –o3 options only, the compiler reverts from the default optimization level (–op2) to –op0. The compiler uses –op0 because it presumes that the calls to the assembly language functions that are called from C may call other C functions or modify C variables.

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

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

- Compile with –pm –o3 –op1.
- Add the volatile keyword to those variables that may be modified by the assembly functions and compile with –pm –o3 –op2. For more information on the volatile keyword, see subsection 4.5.5, *The Volatile Keyword*, on page 4-12.

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

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

- You achieve the best optimization by applying the FUNC_EXT_CALLED pragma to all of the entry-point functions called from the assembly language interrupts, and then compiling with –pm –o3 –op2. *Ensure that you use the pragma with all of the entry-point functions. If you do not, the compiler removes the entry-point functions that are not preceded by the FUNC_EXT_CALLED pragma.*
- The other alternative, which does not require that you apply the FUNC_EXT_CALLED pragma to your entry-point functions, is to compile with –pm –o3 –op3. Because you do not use the pragma, you must use the –op3 option, which is less aggressive than the –op2 option, and your optimization may not be as effective.

Keep in mind that if you use –pm –o3 without additional options, the compiler removes the C functions that the assembly functions call. Use the FUNC_EXT_CALLED pragma to keep these functions.

See subsection 2.4.1.2 for information about the –op*n* option.

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

To see which program-level optimizations the compiler is applying, use the –on2 option to generate an information file. See subsection 2.4.1.4 for more information.

2.4.1.2 Controlling the Aggressiveness of Program-Level Optimization (–op Option)

You can control the aggressiveness of program-level optimization, which you invoke with –pm –o3, by using the –op option. Specifically, the –op option indicates if functions in other modules can call a module's extern functions or modify the module's extern variables. The number following –op indicates the level you set for the module that you are allowing to be called or modified. The –o3 option combines this information with its own file-level analysis to decide whether to treat this module's extern function and variable definitions as if they had been declared static. Use Table 2–3 to select the appropriate level to append to the –op option.

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

If your module …	**Use this option**
Has functions that are called from other modules and global variables that are modified in other modules	–op0
Does not have functions that are called by other modules but has global variables that are modified in other modules	–op1
Does not have functions that are called by other modules or global variables that are modified in other modules	–op2
Has functions that may be called from other modules but does not have global variables that are modified in other modules	–op3

In certain circumstances, the compiler reverts to a different –op level from the one you specified, or it might disable program-level optimization altogether. Table 2–4 lists the combinations of –op levels and conditions that cause the compiler to revert to other –op levels.

Table 2–4. Special Considerations When Using the –op Option

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

In some situations when you use –pm and –o3, you *must* use an –op option or the FUNC_EXT_CALLED pragma. Refer to pages 2-27 to 2-28 for information about these situations.

2.4.1.3 Controlling File-Level Optimizations (–ol Option)

When you invoke the optimizer with the –o3 option, some of the optimizations use known properties of the standard library functions. If your file redefines any of these standard library functions, these optimizations become ineffective. The –ol option (lowercase L) controls file-level optimizations. The number following the –ol denotes the level (0, 1, or 2). Use Table 2–5 to select the appropriate level to append to the –ol option.

Table 2–5. Selecting a Level for the –ol Option

If your file …	**Use this option**
Defines a function with the same name as a standard library function	–ol0
Contains the standard library definition functions for those functions defined in the file	–ol1
Does *not* alter standard library functions, but you have used the –ol0 or the –ol1 option in a command file, an environment variable, etc., and now you want to restore the default behavior of the optimizer	–ol2

2.4.1.4 Creating an Optimization Information File (–on Option)

When you invoke the optimizer with the –o3 option, you can use the –on option to create an optimization information file that you can read. The number following the –on denotes the level (0, 1, or 2). The resulting file has an .nfo extension. Use Table 2–6 to select the appropriate level to append to the –on option.

Table 2–6. Selecting a Level for the –on Option

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

2.4.2 Debugging Optimized Code

When debugging a program, ideally you should debug it in an unoptimized form and reverify its correctness after it has been optimized. You can use the debugger with optimized code, but the optimizer's extensive rearrangement of code and the many-to-one allocation of variables to registers often make it difficult to correlate source code with object code.

Note: Symbolic Debugging and Optimized Code

If you use the –g option to generate symbolic debugging information, many code generator optimizations are disabled because they disrupt the debugger. If you want to use symbolic debugging and still generate fully optimized code, use the –mn shell option; –mn reenables the optimizations disabled by –g. See page 2-14 for more information about the –g option and page 2-15 for more information about the –mn option.

2.4.3 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.

Also, see subsection 4.5.5, *The volatile Keyword,* on page 4-12 for information on preserving necessary memory accesses.

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

2.4.3.2 Use Caution When Accessing Aliased Variables

Aliasing occurs when a single object can be accessed in more than one way, such as when two pointers point to the same object or when a pointer points to a named object. Aliasing can disrupt optimization, because any indirect reference could potentially refer to any other object. The optimizer analyzes the code to determine where aliasing can and cannot occur, then optimizes as much as possible while still preserving the correctness of the program. The optimizer behaves conservatively. If there is a chance that two pointers could be pointing to the same object, then the optimizer assumes that the pointers do point to the same object.

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

2.5 Using Inline Function Expansion

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

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

Inline function expansion is performed one of the following ways:

- Intrinsic operators are expanded by default.
- Automatic inline expansion is performed on small functions that are invoked by the optimizer with the –o3 option.
- Definition-controlled inline expansion is performed when you invoke the compiler with optimization *and* the compiler encounters the inline keyword in code.

2.5.1 Inlining Intrinsic Operators

The compiler automatically expands the intrinsic operators of the target system (such as abs) by default. This expansion 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 the intrinsic operators are:

- abs
- fabs

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

The optimizer inlines small functions when it is invoked with the –o3 option. A command-line option, –oi*size*, controls the size of functions inlined when the optimizer is invoked with the –o3 option. Specify the *size* limit for the largest function that will be inlined. The –oi*size* option can be used two ways:

- If you set the *size* parameter to 0 (–oi0), all size-controlled inlining is disabled.
- If you set the *size* parameter to a nonzero integer, the compiler inlines functions based upon *size*. The optimizer multiplies the number of times the function will be inlined by the size of the function (plus one if the function is externally visible and its definition cannot be safely removed). The optimizer inlines the function only if the result is less than the size parameter. The compiler measures the size of a function in arbitrary units; however, the optimizer information file (created with the –on1 or –on2 option) reports the size of each function in the same units that the –oi option uses.

The –oi*size* option controls only the inlining of functions that have not been explicitly declared as INLINE. If you do not use the –oi option, the optimizer inlines very small functions. The –x shell option controls the inlining of functions declared as INLINE (see subsection 2.5.3).

2.5.3 Controlling Inline Function Expansion (–x Option)

The –x shell option controls the definition of the _INLINE preprocessor symbol and some types of inline function expansion. There are three levels of expansion:

–x0 causes no definition-controlled inline expansion. This option overrides the default expansions of the intrinsic operator functions, but it does not defeat the inline function expansions described in subsection 2.5.2 on page 2-35.

–x1 resets the default behavior. The intrinsic operators (abs, labs, and fabs) are inlined wherever they are called. Use this option 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** defines the _INLINE preprocessor symbol to be 1, and, if the optimizer is not invoked with a separate command-line option, invokes the optimizer at the default level (–o2).

2.5.4 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, with the exception of functions declared as static inline. In functions defined as static inline, the expansion occurs despite the presence of local statics. In addition, a limit is placed on the depth of inlining for recursive or non-leaf functions. Inlining should be used for small functions or functions that are called in a few places (though the compiler does not enforce this). You can control this type of function inlining with the inline keyword.

The inline keyword specifies that a function is expanded inline at the point at which it is called rather than by using standard calling procedures. You can use the inline keyword two ways:

- By *defining* a function as inline within a module
- By *declaring* a function as static inline

2.5.4.1 Using the inline Keyword to Define a Function as Inline Within a Module

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. It is called by other modules unless the modules contain a compatible static inline declaration. Functions defined as inline are expanded when the optimizer is invoked and the –x0 option has not been specified on the command line. Using the –x2 option automatically invokes the optimizer at the default level (level2). Use this syntax to define a function as inline within a module:

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

2.5.4.2 Using the inline Keyword to Declare a Function as Static Inline

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). This names the function and specifies that the function is to be expanded inline, but no code is generated for the function declaration itself. A function declared in this way can be placed in header files and included by all source modules of the program. Declaring a function as static inline in a header file specifies that the function is inlined in any module that includes the header.

Functions declared as inline are expanded whenever the optimizer is invoked at any level. Functions declared as inline and controlled by the _INLINE preprocessor symbol, such as the runtime-library functions, are expanded whenever the optimizer is invoked and the _INLINE preprocessor symbol is equal to 1. When you define an inline function, it is recommended that you use the _INLINE preprocessor symbol to control its declaration. If you fail to control the expansion using _INLINE and subsequently compile *without* the optimizer, the call to the function will be unresolved. For more information on the _INLINE preprocessor symbol, see subsection 2.5.5.

Use this syntax to declare a function as static inline:

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

Note: Function Inlining Can Greatly Increase Code Size

Expanding functions inline expands code size, and inlining a function that is called a great number of times will characteristically increase code size. Function inlining is optimal for functions that are called only a small number of times and for small functions that are called more often. If your code size seems too large, try compiling with the –x0 option and note the difference in code size.

2.5.5 The _INLINE Preprocessor Symbol

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 runs 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 2–1 on page 2-38 illustrates how the runtime-support library uses the _INLINE preprocessor symbol.

The _INLINE preprocessor symbol is used in the string.h header file to declare the function correctly, regardless of whether inlining is used. The _INLINE preprocessor symbol is used to conditionally define _ _INLINE so that strlen is declared as static inline only if the _INLINE preprocessor 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 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 the same way that the function libraries use it so that your programs run regardless of whether inlining is selected for any or all of the modules in your program.

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

(a) string.h

```
/*****************************************************************************/
/* string.h  v6.xx                                                           */
/* Copyright (c) 1995 Texas Instruments Incorporated                         */
/*****************************************************************************/

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

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

_ _INLINE size_t  strlen(const char *s);

#ifdef _INLINE
```

(b) strlen.c

```
/*****************************************************************************/
/*  strlen                                                                   */
/*****************************************************************************/
static inline size_t strlen(const char *s)
{
   const char *r = s – 1;

   while (*++r);
   return r – s;
}

#endif

#undef _ _INLINE

#endif
```

In Example 2–1, there are two definitions of the strlen function. The first, in the header file, 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, for the library, ensures that the callable version of strlen exists when inlining is disabled. Since this is not an inline function, the _INLINE preprocessor symbol is undefined (#undef) before string.h is included to generate a noninline version of strlen's prototype.

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

2.6 Using the Interlist Utility

The compiler tools include a utility that interlists C source statements into the assembly language output of the compiler. The interlist utility enables you to inspect the assembly code generated for each C statement. The 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 –s shell option. To compile and run the interlist utility on a program called function.c, enter:

```
cl370 -s function
```

The –s option prevents the shell from deleting the interlisted assembly language file (as if you had used the –k option). The output assembly file, function.asm, is assembled normally.

2.6.1 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 ;>>>>. The interlist utility may affect optimized code because it may prevent some optimization from crossing C statement boundaries.

Example 2–2 shows a typical interlisted assembly file.

Example 2–2.An Interlisted Assembly Language File

```
;>>>>  main()
;***************************************************************
;
; FUNCTION DEF: _main
;***************************************************************
;
_main:
;>>>>      i+=j;
         ADD       _j,_i
;>>>>      j=i+123;
         MOV       _i,_j
         ADD       #123,_j
         RTS
   .globreg   _i
   .reg       _i,1
   .globreg   _j
   .reg       _j,1
```

2.6.2 Using the Interlist Utility With the Optimizer

If the –s option is used with the optimizer (the –o*n* option), 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 ;**.

To view the optimizer comments and the original C source interlisted with the assembly code, use the –ss option with the –o option. With the –ss option, the optimizer inserts its comments and the interlist utility runs between the code generator and the assembler, merging the original C source into the assembly file.

Example 2–3 shows the function from Example 2–2 compiled with the optimizer (–o2) and the –s option.

Example 2–3. The Function From Example 2–2 Optimized

```
;   opt370 –s –O2 main.if main.opt
;***********************************************************
;* FUNCTION DEF: _main
;***********************************************************
_main:
;*** 4  ----------------------    i += j;
         ADD       _j,_i
;*** 5  ----------------------    j = (int)i+123;
         MOV       _i,_j
         ADD       #123,_j
;***    ----------------------    return;
         RTS


    .globreg   _i
    .reg       _i,1
    .globreg   _j
    .reg       _j,1
```

2.7 Understanding and Handling Compiler Errors

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

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

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

Errors in C code are divided into 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. Examples of each level of error message are located in Table 2–7.

- **Code-W errors** are warnings: they result from a condition that is technically undefined according to the rules of the language or that can cause unexpected results.
- **Code-E errors** are recoverable: they result from a condition that violates the semantic rules of the language. Although these are normally fatal errors, the compiler can recover and generate an output file if you use the –pe option. Refer to subsection 2.7.1 for more information.
- **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.
- **Code-I errors** are implementation errors: they occur when one of the compiler's internal limits is exceeded. These errors are usually caused by extreme behavior in the source code rather than by explicit errors. In most cases, code-I errors cause the compiler to abort immediately. Most code-I messages contain the maximum value for the limit that was exceeded. (Those limits that are absolute are also listed in Section 4.10, *Compiler Limits*, on page 4-20.)
- **Other error messages**, such as incorrect command line syntax or inability to find specified files, are usually fatal. They are identified by the symbol >> preceding the message.

Table 2–7. Example Error Messages

Error level	Example error message
Code W	`"file.c", line 42:[W029] extra text after preprocessor directive ignored`
Code E	`"file.c", line 66: [E055] illegal storage class for function 'f'`
Code F	`"file.c", line 71: [F0108] structure member 'a' undefined`
Code I	`"file.c", line 99: [I011] block nesting too deep (max=20)`
Other	`>> Cannot open source file 'mystery.c'`

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

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

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.

Refer to subsection 2.7.3 for an example of the –pe option.

2.7.2 Altering the Level of Warning Messages (–pw Option)

You can determine which levels of warning messages (code-W error messages), if any, 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 2–8 to select the appropriate level to append to the –pw option. (The –pw1 option is the default.)

Table 2–8. Selecting a Level for the –pw Option

If you want to…	Use this 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.	–pw1
Enable all warning messages.	–pw2

Refer to subsection 2.7.3 for an example of the –pw option.

2.7.3 An Example of How You Can Use Error/Warning Options

The following example demonstrates how you can suppress errors with the –pe option and/or alter the level of error messages with the –pw option. The examples use this code segment contained in file err.c:

```
   int *pi; char *pc;

#if STDC
   pi = (int *) pc;
#else
   pi = pc;
#endif
```

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

  ```
  [err.c]
  "err.c", line 8: [E122] operands of '=' point to different types
  ```

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

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

  ```
  [err.c]
  "err.c", line8: [E122] 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 –pew2 (combining –pe and –pw2), this is the result:

  ```
  [err.c]
  "err.c", line5: [W038] undefined preprocessor symbol 'STDC'
  "err.c", line8: [E122] operands of '=' point to different types
  ```

  As in the previous case, –pe causes the compiler to overlook the error and generate code. Because the –pw2 option is used, all warning messages are generated.

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 method is especially useful when you have multiple source files.
- ❏ You can compile and link in one step by using cl370. This method is useful when you have a single source module.

This chapter describes how to invoke the linker with each method. It also discusses special requirements of linking C code, including the runtime-support libraries, specifying the initialization model, and allocating the program into memory. For a complete description of the linker, see the *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide*.

Topic **Page**

3.1 Invoking the Linker as an Individual Program

This section shows 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:

lnk370 {**–c** | **–cr**} *filenames* [**–o** *name.out*] **–l** *libraryname* **lnk.cmd**

lnk370 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 lnk370, you must use –c or –cr. The –c option uses ROM initialization; the –cr option uses RAM initialization.

filenames are object files created by compiling C programs or assembling assembly language programs. If you have specified the files in a linker command file, you do not need this parameter.

–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* (lowercase L) identifies the appropriate archive library containing C runtime-support and floating-point math functions. If you have specified a runtime-support library in a linker command file or the library is in the current directory, you do not need this parameter. (The –l option tells the linker to use C_DIR for the search path to find an archive path or object file.) If you're linking C code, you must use either the library included with the compiler or your own runtime-support library that you have created. For more information on the runtime-support library and its contents, see Section 6.1, *Libraries*, on page 6-2.

lnk.cmd is the linker command file used to link a C program. This file is optional. For more information, see subsection 3.4.5 on page 3-10.

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):

```
lnk370 –c prog1 prog2 prog3 –o prog.out –l rts.lib
```

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. For more information, see the *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide*.

3.2 Invoking the Linker With the Compiler Shell (–z Option)

The options and parameters discussed in this section apply to both methods of linking; however, when you are linking with the shell, the options must follow the –z shell option (see the syntax shown on page 2-3).

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:

- The –z option must follow all source files and compiler options on the command line (or be specified with the C_OPTION environment variable).
- The –z option divides the command line into compiler options (the options before –z) and linker options (the options following –z).
- The –c shell option suppresses –z, so do not use the –c shell option before the –z option on the shell's command line if you want to link. For more information on the –c shell option, see the subsection *Disabling the Linker (–c Shell Option)* on page 3-4. You can use –c following –z on the command line as a valid linker option. See section 3.4.2 on page 3-8 for more information.

All arguments that follow –z on the command line are passed to the linker. These arguments can be linker command files, additional object files, linker options, or libraries. For example, to compile and link all the .c files in a directory, enter:

```
cl370 –sq *.c –z c.cmd –o prog.out –l rts.lib
```

First, the shell compiles all of the files in the current directory that have a .c extension and uses the –s and –q options. Second, because the –z option is specified, the linker links the resulting object files by using the c.cmd command file; the –o option names the output file, and the –l option names the runtime-support library.

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

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

Disabling the Linker (–c Shell Option)

You can override the –z option by using the –c shell option. This option is especially helpful when you have specified the –z option in the C_OPTION environment variable and want to selectively disable linking with the –c shell option 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 uses the –c linker option when you use the –z option. This 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 following the –z option.

3.3 Linker Options

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

–a produces an absolute, executable module. This is the default; if neither –a nor –r is specified, the linker acts as if –a is specified.

–ar produces a relocatable, executable object module.

–c uses linking conventions defined by the ROM autoinitialization model of the TMS370/C8 8-bit C compiler.

–cr uses linking conventions defined by the RAM autoinitialization model of the TMS370/C8 8-bit 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* declares a *global_symbol* as global even if the global symbol has been made static with the –h option.

–h makes all global symbols static.

–hstack*size* sets the C hardware stack size to *size* bytes and defines a global symbol that specifies the stack size. Default = 32 bytes.

–heap *size* sets heap size (for the dynamic memory allocation in C) to *size* bytes and define a global symbol that specifies the heap size. Default = 512 bytes.

–i *directory* 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.

–j disables conditional linking.

–l *filename* names an archive library file as linker input; *filename* is an archive library name. The filename must follow operating system conventions.

–m *filename* produces a map or listing of the input and output sections, including holes, and place the listing in *filename.* The filename must follow operating system conventions.

–n	ignores all fill specifications in memory directives.
–novy	avoids processing of overlay sections and disables automatic overlay of register and memory blocks.
–o *filename*	names the executable output module. The default filename is a.out. The filename must follow operating system conventions.
–q	requests a quiet run (suppress the banner).
–r	retains relocation entries in the output module.
–s	strips symbol table information and line number entries from the output module.
–sstack *size*	sets the C software stack size to *size* bytes and defines a global symbol that specifies the stack size. Default=0 bytes.
–u *symbol*	places the unresolved external symbol *symbol* into the output module's symbol table.
–w	display a message when an undefined output section is created.
–x	forces rereading of libraries. Resolves back references.

For more information on linker options, refer to the *Linker Description* chapter of the *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide.*

3.4 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 the *Linker Description* chapter of the *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide.*

3.4.1 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. If your rts.lib (rts_c8.lib for the TMS370C8) is in the current directory, you do not need to use any options to link to it. If rts.lib (or rts_c8.lib) is not in the current directory, use the –l option to tell the linker to look in C_DIR to find an archive path or object file. To use the –l option, type on the command line:

lnk370 {**–c** | **–cr**} *filenames* **–l** *libraryname*

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

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

One important function contained in the runtime support library is _c_int00. The symbol _c_int00 is the starting point in boot.obj; if you use the –c or –cr option, then _c_int00 is automatically defined as the entry point for the program. If your program begins running from reset, you should set up the reset vector to branch to _c_int00 so that the processor executes boot.obj first.

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

- Switches to user mode and sets up the user mode stack
- Processes the runtime initialization table and autoinitializes global variables (in the ROM model)
- Calls main
- Calls exit when main returns

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

3.4.2 Specifying the Initialization Model (RAM and ROM)

The C compiler produces tables of data for autoinitializing global variables. Subsection 5.9.1, *Autoinitialization of Variables and Constants*, on page 5-37 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 are initialized at *load time*; use the –cr linker option to select the RAM model. For more information about the RAM model, refer to page 5-39.

- *ROM model* (–c linker option)

  Global variables are initialized at *runtime*; use the –c linker option to select the ROM model. For more information about the ROM model, refer to page 5-40.

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; for example, they tell the linker to select the ROM or RAM model of autoinitialization. When you use the shell to link programs, the –c option is the default (–c must follow –z when invoking the linker through the shell. See subsection 3.2, page 3-3 for more information.). The following list outlines the linking conventions used with –c or –cr:

- The symbol _c_int00 is defined as the program entry point; it identifies the beginning of the C boot routine in boot.obj. When you use –c or –cr, _c_int00 is automatically referenced; this ensures that boot.obj is automatically linked in from the runtime-support library.
- 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 (the –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 (010h) is set in the .cinit section header. STYP_COPY is the special attribute that tells the loader to perform autoinitialization directly and not to load the .cinit section into memory. The linker does not allocate space in memory for the .cinit section.

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

3.4.3 Specifying Where to Allocate Sections in Memory

The compiler produces relocatable blocks of code and data. These blocks, called *sections*, can be allocated into memory in a variety of ways to conform to a variety of system configurations.

The compiler creates two basic kinds of sections: initialized and uninitialized. Table 3–1 summarizes the sections.

Table 3–1. Sections Created by the Compiler

Name	Type	Contents
.bss	Uninitialized	Global and static variables
.cinit	Initialized	Tables for explicitly initialized global and static variables
.const	Initialized	Global and static const variables that are explicitly initialized and string literals
.hstack	Uninitialized	Hardware stack
.sstack	Uninitialized	Software stack
.sysmem	Uninitialized	Memory for malloc functions
.text	Initialized	Executable code and constants

When you link your program, you must specify where to allocate the sections in memory. In general, initialized sections can be linked into ROM or RAM; uninitialized sections must be linked into RAM. Refer to subsection 5.1.1, *Sections*, on page 5-2 for a complete description of how the compiler uses these sections. The linker provides MEMORY and SECTIONS directives for allocating sections. For more information about allocating sections into memory, refer to the *Linker Description* chapter of the *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide*.

3.4.4 Specifying Where to Allocate Subsections in Memory

Subsections allow you to manipulate sections on a deeper level. You can specify subsections in the linker command file's SECTIONS directive to have the directive apply to part of a section.

A section defined with a subsection name does have to be accounted for explicitly in the linker command file. If you are not concerned about the placement of a subsection, you can still manipulate the subsection using the subsection's base name. This means that if you do not specify where to allocate a specific subsection, then the subsection will be included with all of the sections that bear the same base name (the name of the section that is specified before the : in the definition of the subsection).

For a complete discussion on subsections, see subsection 5.1.2, *Subsections*, on page 5-4.

3.4.5 A 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 lnk.cmd (lnkc8.cmd for the TMS370C8) and lists several linker options:

–c	is one of the options that can be used to link C code; it tells the linker to use the ROM autoinitialization method.
–hstack 020h	tells the linker to set the C hardware stack size at 0x0200 bytes.
–sstack 200h	tells the linker to set the C software stack size at 0x0200 bytes
–heap 200h	tells the linker to set the heap size to 0x0200 bytes.
–lrts.lib	tells the linker to use an archive library file, rts.lib, for input.

To link the program, enter:

lnk370 *object_file(s)* **–o** *outfile* **–m** *mapfile* lnk.cmd

The MEMORY directive, and possibly the SECTIONS directive, may require modification to work with your system. See the *Linker Description* chapter of the *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide* for information on these directives.

Example 3–1.Linker Command File for the TMS370

```
/*************************************************************************/
/*  LNK370.CMD – COMMAND FILE FOR LINKING 370 C PROGRAMS                 */
/*************************************************************************/
–c                                        /* LINK USING C CONVENTIONS      */
–hstack 020h                              /* 32 BYTES OF HARDWARE STACK    */
–sstack 200h                              /* 512 BYTES OF SOFTWARE STACK   */
–heap   200h                              /* 512 BYTES OF HEAP AREA        */
–lrts.lib                                 /* INCLUDE ANSI LIBRARY          */

/*************************************************************************/
/* SPECIFY THE SYSTEM MEMORY MAP                                         */
/*************************************************************************/
MEMORY
{
   RFILE        : org =  000eh         len =  00f2h /* UNUSED REGISTER FILE */
   D_MEM        : org =  2000h         len =  2000h /* DATA MEMORY (RAM)    */
   P_MEM        : org =  4000h         len =  3fech /* PROGRAM MEMORY (ROM) */
   TVEC_MEM     : org =  7fc0h         len =  0020h /* TRAP VECTOR MEM      */
   IVEC_MEM     : org =  7fech         len =  0014h /* INTERRUPT VEC MEM    */
}

/*************************************************************************/
/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY                           */
/*************************************************************************/
SECTIONS
{
   .reg      : {} > RFILE              /* NEAR GLOBAL & STATIC VARS         */
   .regovy   : {} > RFILE              /* REGISTER OVERLAY PARTITIONS       */
   .hstack   : {} > RFILE              /* HARDWARE STACK AREA               */
   .bss      : {} > D_MEM              /* FAR GLOBAL & STATIC VARS          */
   .bssovy   : {} > D_MEM              /* MEMORY OVERLAY PARTITIONS         */
   .heap     : {} > D_MEM              /* DYNAMIC MEMORY ALLOCATION AREA    */
   .sstack   : {} > D_MEM              /* SOFTWARE STACK AREA               */
   .text     : {} > P_MEM              /* CODE                              */
   .cinit    : {} > P_MEM              /* INITIALIZATION TABLES             */
   .const    : {} > P_MEM              /* CONSTANT DATA                     */
   .trapvecs : {} > TVEC_MEM           /* TRAP VECTORS                      */
   .intvecs  : {} > IVEC_MEM           /* INTERRUPT VECTORS                 */
}
```

Chapter 4

TMS370/C8 C Language

The C language that the TMS370/C8 8-bit C compiler supports is based on the 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 ANSI standard is described in the American National Standard for Information Systems—Programming Language C X3.159–1989. The second edition of *The C Programming Language* is based on the ANSI standard and is a reference. ANSI C encompasses many of the language extensions provided by current C compilers and formalizes many previously unspecified characteristics of the language.

The TMS370/C8 8-bit C compiler follows the ANSI C standard. The ANSI standard identifies certain implementation-defined features that may differ from compiler to compiler, depending on the type of processor, the runtime environment, and the host environment. This chapter describes how these and other features are implemented for the ’370/C8 8-bit C compiler.

Topic **Page**

4.1 Characteristics of TMS370/C8 C

The ANSI standard identifies certain features of the C language; these features are affected by characteristics of the target processor, runtime environment, or host environment, which, for reasons of efficiency or practicality, may differ among standard compilers. This section describes how these features are implemented for the '370/C8 8-bit C compiler.

The following list identifies all such cases and describes the behavior of the '370/C8 8-bit C compiler in each case. Each description also includes a reference to the formal ANSI standard and to the *The C Programming Language* by Kernighan and Ritchie (K&R).

4.1.1 Identifiers and Constants

- The first 200 characters of all identifiers are significant. Case is significant; uppercase and lowercase characters are distinct in identifiers. These characteristics apply to all identifiers, internal and external, in all '370/C8 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)

- Hex or octal escape sequences in character or string constants may have values up to 8 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

- For information about the representation of data types, refer to Section 4.2. (ANSI 3.1.2.5, K&R A4.2)

- The type size_t, which is the result of the sizeof operator, is unsigned int. (ANSI 3.3.3.4, K&R A7.4.8)

- The type ptrdiff_t, which is the result of pointer subtraction, is int. (ANSI 3.3.6, K&R A7.7)

4.1.3 Conversions

- Float-to-int conversions truncate toward zero.
  (ANSI 3.2.1.3, K&R A6.3)
- Shortening a signed integer truncates it (keeps lower bits).
  (ANSI 3.2.1.2, K&R A6.2)
- Pointers and integers can be freely converted.
  (ANSI 3.3.4, K&R A6.6)

4.1.4 Expressions

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

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

4.1.5 Declarations

- The *register* storage class is effective for all arithmetic types and for all pointers. For more information, refer to Section 4.3.
  (ANSI 3.5.1, K&R A8.1)
- Structure members are not packed into bytes (with the exception of bit fields). Each member is aligned on an 8-bit boundary.
  (ANSI 3.5.2.1, K&R A8.3)
- A bit field defined as an integer is treated as an unsigned quantity. A bit field cannot be defined as a signed integer. Bit fields are packed into bytes beginning at the high-order bits and do not cross byte boundaries. Therefore, bit fields are limited to a maximum size of 8 bits, regardless of what type is used in the C source. (ANSI 3.5.2.1, K&R A8.3)

4.1.6 Runtime Support

- The following library is included in addition to the libraries specified by the ANSI C standard. See Section 6.1 for information on the header files that correspond with this library.
  - ports.h (see subsection 6.1.6, page 6-6)
- The following ANSI C runtime support functions are not supported:
  - locale.h
  - signal.h
  - time.h

  (ANSI 4.1, K&R B)
- The stdlib library functions getenv and system are not supported.

  (ANSI 4.10.4, K&R B5)
- For functions in the math library that produce a floating-point return value, if the value is too small to be represented, zero is returned and errno is set to ERANGE. See subsections 6.1.3 on page 6-4 and 6.1.5 on page 6-6 for more information on error reporting and floating-point math.

  (ANSI 4.5.1, K&R B4)

4.2 Data Types

Table 4–1 lists the size, representation, and range of each scalar data type. 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, refer to subsection 6.1.4, page 6-4.

Table 4–1. TMS370/C8 C Data Types

Type	Size	Representation	Range	
			Minimum	Maximum
char, signed char	8 bits	ASCII	–128	127
unsigned char	8 bits	ASCII	0	255
short	16 bits	2s complement	–32 768	32 767
unsigned short	16 bits	binary	0	65 535
int, signed int	16 bits	2s complement	–32 768	32 767
unsigned int	16 bits	binary	0	65 535
long, signed long	32 bits	2s complement	–2 147 483 648	2 147 483 647
unsigned long	32 bits	binary	0	2^{32}–1
enum	16 bits	2s complement	–32 768	32 767
float	32 bits	IEEE 32-bit	–1.18e–38	3.40e+38
double	32 bits	IEEE 32-bit	–1.18e–38	3.40e+38
long double	32 bits	IEEE 32-bit	–1.18e–38	3.40e+38
pointers	16 bits	binary	0	0xFFFF

4.3 Register Variables

The TMS370/C8 8-bit C compiler treats register variables (variables declared with the register keyword) differently, depending on whether or not you use the optimizer.

- **Compiling with the optimizer**

  The compiler ignores any register declarations and allocates registers to variables and temporary values by using a cost algorithm that attempts to make the most efficient use of registers.

- **Compiling without the optimizer**

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

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

For more information about register variables, refer to subsection 5.3.1, page 5-17.

4.4 Port Registers

The TMS370/C8 8-bit C compiler extends the C language by adding a naming convention to reference the ’370/C8’s ports. This naming convention allows you to define and use a name that the compiler automatically maps directly to one of the ’370/C8 port registers. The compiler can then use special port operands to generate assembly language instructions. You can name a peripheral register and manipulate the bits within the register using structure references using the PORT pragma. For more information on using the PORT pragma, see subsection 4.6.2, *The PORT Pragma*, on page 4-14.

4.4.1 Referencing Registers With the Naming Convention

You can reference a particular port by prefixing either the decimal or hexadecimal number of the port with an underscore and a capital P. For example, to reference port 10, you can use _P10 (decimal name) or _P0A (hexadecimal name).

If the numeric portion of the name begins with the digit 0, then the number is interpreted as a hex value; otherwise, it is interpreted as a decimal value. If the numeric portion of the name represents a value outside the range of 0–255, the name is treated as a regular identifier.

No storage is allocated for port names; the storage is implied to be the port registers associated with the names. Example 4–1 illustrates the use of port names.

Example 4–1. Using Port Registers in Functions

This section of a function:

```
init()
{
    .
    .
    .
   extern volatile unsigned char _P050 = 0x77;
   extern volatile unsigned char _P051 = 0x33;
   extern volatile unsigned char _P059 = 0x00;
   _P059 &= 31;
}
```

generates this ’370/C8 8-bit assembly language code:

```
MOV   #077h, P050
MOV   #033h, P051
MOV   #00h,  P059
AND   #31,   P059
```

The file ports.h supplied with the compiler defines all the ports as extern volatile unsigned char and equates descriptive names to the port names. The descriptive names are equivalent to those used in the *TMS370 Family User's Guide* or the *TMS370C8 Central Processing Unit, System, and Instruction Set Reference Guide.* Refer to subsection 6.1.6, page 6-6, for more information about ports.h.

4.4.2 Binding Structure Definitions to Peripheral Registers

The compiler allows you to associate variables that are bound to peripheral registers by the PORT pragma with structure types or any other integral type. Using structure references, you can name a peripheral register and manipulate the bits within the register.

Use the PORT pragma to associate a variable with a port. The variable can be any integral type or structure containing integral type fields. If a variable that has been bound to a port register is larger than a byte, then subsequent bytes in the variable are associated with subsequent port registers. Example 4–2 illustrates how to declare and reference a peripheral register that has a structure bound to it.

Example 4–2. Declaring and Referencing a Register Bound to a Structure

```
#pragma PORT (pstr, P011)

volatile struct {
         unsigned int    fld1:3;
         unsigned int    fld2:2;
         unsigned int    fld3:3;
}pstr;

void foo() {pstr.fld3 = 2;}
```

4.5 Keywords

The TMS370/C8 8-bit C compiler supports the const keyword, the interrupt keyword, the reentrant keyword, the near and far keywords, and the volatile keyword.

4.5.1 The const Keyword

The TMS370/C8 8-bit C compiler supports the ANSI standard keyword *const*. This keyword allows you to have greater control over allocation of storage for certain data objects. The const qualifier can be applied to the declaration of any variable to specify that its value will not be changed. For an array, the const qualifier says that the elements will not be altered.

If an object is declared as const, then storage for the object will be allocated from the .const section, which can be linked into ROM. There are two exceptions to the const data storage allocation rule. In both cases, the storage for the object will be the same as if the const keyword was not used. The exceptions are:

- If the volatile keyword is also specified in the declaration of an object. Volatile keywords are assumed to be far and are allocated to RAM. (The program will not modify a const volatile object, but something external to the program might.)
- If the object is auto (function scope)

The placement of the const keyword within a declaration can be important. For example,

```
int * const p = &x;
```

declares a constant pointer p to a variable int, and

```
const int * q = &x;
```

declares a variable pointer q to a constant int.

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

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

4.5.2 The interrupt Keyword

The ’370/C8 C compiler extends the C language by adding the interrupt keyword to specify that a function is to be treated as an interrupt function.

Functions that handle interrupts follow special register saving rules and a special return sequence. When C code is interrupted, the interrupt routine must preserve the contents of all machine registers that are used by the routine or by any function called by the routine. When you use the interrupt keyword with the definition of the function, the compiler generates register saves based on the rules for interrupt functions and the special return sequence for interrupts.

The interrupt keyword can be used only with a function that is defined to return void and that has no parameters. The body of the interrupt function can have local variables and is free to use the stack. For example:

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

    ...
}
```

Unless specified otherwise by the INTERRUPT pragma, the interrupt keyword defines the function as an IRQ (interrupt request) interrupt type.

4.5.3 The reentrant Keyword

The reentrant keyword provides you with the capability to declare a function reentrant:

```
reentrant int fact (int n)
{
      if (n==1) return 1;
      return (n * fact(n–1));
}
```

The above factorial function receives its parameter *n* in register pair R2:R3. On entry into the function, space is allocated on the software stack for *n*, and the value of *n* is moved from register pair R2:R3 to the software stack. The value in R2:R3 is decremented and passed to the next activation of fact(). Prior to exit, the return value is placed in register pair R2:R3, and the stack space allocated to *n* is deallocated.

4.5.4 The near and far Keywords

The near and far keywords are used to specify storage allocation for global, static, and local (or automatic) variables:

near keyword allocates objects in the on-chip register file of the TMS370/C8, allowing quick access.

far keyword allocates objects in other RAM memory, allowing for larger data sizes.

Scalar and structure variables are, by default, assumed to be near the compiler and are allocated in the register file unless otherwise directed. Arrays are, by default, assumed to be far and are allocated to an uninitialized memory section (such as .bss).

If a variable is defined without a near or far keyword or pragma associated with it then it follows these rules in terms of where it is placed:

Table 4–2. Data Placement Rules

Data Type	Location
auto scalar (default)	expression registers / overlay register block (.reg)
auto struct (default)	overlay register block (.reg)
auto array (default)	overlay memory block (.bss)
auto array declared near	overlay register block (.reg)
auto scalar or struct declared far	overlay memory block (.bss)
auto in reentrant function (always)	software stack
static/extern scalar	fixed register block (.reg)
static/extern struct	fixed register block (.reg)
static/extern array	fixed memory block (.bss)
static/extern declared near	fixed register block (.reg)
static/extern declared far	fixed memory block (.bss)

For more information about overlay blocks, see Section 5.4, *Overlaying Automatic Variables*, on page 5-19.

Syntactically, near and far keywords are treated as storage class modifiers. The near and far keywords can appear before, after, or in between the specifiers of storage class and types. For example,

```
near  static int x;
static near  int x;
static int   far x;
```

Near and far keywords can be used for:

- Declarations containing a static storage class keyword
- Declarations containing an extern storage class keyword
- Declarations in file scope (an external definition) that have no class specifier
- Declarations that are automatic

For example,

```
int plain;                /* these are file scope */
far int f1
;
extern near int n1;
far static int f2;
main()
{
   static near int n2;          /* this is function scope*/
}
```

In the example above, regardless of the memory model, f1 and f2 are allocated in RAM, and n1, n2, and plain are allocated in the internal register file.

Note that storage class modifiers are meaningful only for object declarations, not for functions. Two storage class modifiers cannot be used together in a single declaration.

4.5.5 The volatile Keyword

The optimizer analyzes data flow to avoid memory accesses whenever possible. If you have code that *depends* on memory accesses exactly as written in the C code, you *must* use the volatile keyword to identify these accesses. A variable qualified with a volatile keyword will be allocated to an uninitialized section (as opposed to the register file). The compiler won't optimize out any references to volatile variables.

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

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

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

```
volatile unsigned int *ctrl
```

*ctrl is usually a hardware location, such as an interrupt flag.

4.6 Pragma Directives

The ’370/C8 C compiler supports the following pragmas:

#pragma DATA_SECTION (*symbol*, “*section name*”)
#pragma PORT (*variable, peripheral register name*)
#pragma NEAR (*func*)
#pragma FAR (*func*)
#pragma INTERRUPT (*func[, interrupt_type]*)
#pragma OVLY_IND_CALLS (*calling function, sym1, sym2, ..., symN*)
#pragma CALLS (*func1, func2*)
#pragma TRAP (*func, trap_vector_slot_number*)
#pragma REENTRANT (*func*)
#pragma FUNC_EXT_CALLED (*func)*

The arguments *func* and *symbol* must have file scope; that is, they cannot be defined or declared inside the body of a function. The pragma itself must be specified outside the body of a function and must occur before any declaration, definition, or reference to the symbol or *func* argument.

4.6.1 The DATA_SECTION Pragma

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

#pragma DATA_SECTION (*symbol*, “*section name*”)

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

Example 4–3. 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
```

4.6.2 The PORT Pragma

The PORT pragma allows you to associate a variable with a port. The variable can be any integral type or structure containing integral type fields. The TMS370/C8 compiler allows you to associate variables that are bound to peripheral registers with the PORT pragma with structure types. You can name a peripheral register and manipulate the bits within the register using structure references, as shown in Example 4–4.

Example 4–4. Declaring and Referencing a Peripheral Register Bound to a Structure

```
#pragma PORT (pstr,P011)

volatile struct {
          unsigned int        fld1:3;
          unsigned int        fld2:2;
          unsigned int        fld3:3;
} pstr;
void foo () { pstr.fld3 = 2; }
```

4.6.3 The NEAR and FAR Pragmas

The NEAR and FAR pragmas offer an alternative to using the near and far keywords. Refer to subsection 4.5.4 on page 4-11 for more information on the distinction between near and far data. The following shows how the NEAR and FAR pragmas can be used:

```
#pragma NEAR (statvar1)
static long statvar1;
#pragma FAR (statvar2);
static long statvar2;
```

statvar1 will be allocated into the .reg section and statvar2 will be allocated into the .bss section. The NEAR and FAR pragmas have an advantage over the near and far keywords in that the pragmas are not syntactically attached to the variables that they are qualifying, the pragmas can appear anywhere within the same compilation unit as the variables they affect.

4.6.4 The INTERRUPT Pragma

The INTERRUPT pragma allows you to handle interrupts directly with C code. The argument *func* is the name of a function, and the argument *interrupt_type* is an interrupt type.

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

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.

4.6.5 The OVLY_IND_CALLS Pragma

The OVLY_IND_CALLS pragma specifies a set of functions that can be called indirectly from a specified calling function. The OVLY_IND_CALLS pragma must be specified outside the function definition and must have a function as its first argument:

#pragma OVLY_IND_CALLS (*calling function, sym1, sym2, ..., symN*)

4.6.6 The CALLS Pragma

The CALLS pragma allows you to identify the functions that call another function at some point in time. This example indicates that function f() calls function g() at some point in time:

#pragma CALLS (f,g)

The compiler will recognize the pragma and generate a .calls directive at the point at which the pragma occurs. For more information on the .calls directive, see the *Assembler Directives* chapter in the *TMS370 and TMS370C8 8-Bit Microcontroller Assembly Language Tools User's Guide*.

The CALLS pragma is not syntactically restricted in any way, because it emits a directive. It can be used with the –mi option to provide information to the linker about ancestor and descendant relationships between functions that call one another while avoiding the generation of the .icalls directive in a function that contains an indirect call. For more information on –mi option, see Section 2.2.1, *Frequently Used Command-Line Options,* on page 2-15.

4.6.7 The TRAP Pragma

The TRAP pragma allows you to associate function names with specific trap vectors. If a function is associated with a trap vector, then the compiler replaces calls to that function with a TRAP instruction that directs the processor to use the address stored at the appropriate trap vector slot as the destination address.

A trap instruction is similar to a call with the added benefit of reducing your code size by two bytes per call of the function. Since you can assign only sixteen functions to the sixteen available trap vector slots (0–15), use the TRAP pragma with those functions that are called most often. This will maximize the reduction in code size. Example 4–5 shows a bookkeeping function that is declared as TRAP called several times during the execution of a state machine:

Example 4–5. Declaring a Function as TRAP

```
#pragma TRAP (money, 12)

lotsof (int state)
{
      ...
      state = money (x, y, 1);
      ...
      state = money (23, y + 10, 2);
      ...
      state = money (x – 10, y – 1, 3);
      ...
}

int money (int xpos, int ypos, int transition)
{
      ...
}
```

Each call to money is replaced with a TRAP 12 instruction, where 12 is the trap vector slot associated with the function money. As shown in the example, a trap function can return values or accept arguments just like any other C function.

After you determine which functions you want to declare as TRAP, you need to place the address of each function in the appropriate trap vector slot by creating a table of addresses in the trap vector section. To map trap routines to trap vectors you must update the trapvecs.asm file to include your trap routines:

1) Add a .global directive for the name of your routine to the beginning of the file.
2) Replace the 0 with the name of your routine in the appropriate .word directive. The comments in the .trapvecs.asm file indicate which .word directive is associated with which trap vector. Example 4–6 shows the trapvecs.asm file for Example 4–5. If you are using C trap functions, remember to prefix the name of these functions with an underscore.
3) Assemble and link the trapvecs.asm file with your application's code. You must link this file with a link command file that contains a SECTION directive that maps the .trapvecs section into the appropriate memory location for your target processor ($0\times7fc0$–$0\times7fe0$ for the TMS370 or $0\times7ffc0$–$0\times7ffe0$ for the TMS370C8). The link command file (lnk370.cmd or lnkc8.cmd) supplied with the compiler contains the proper SECTION directives.

Example 4–6.Mapping Trap Routines to Trap Vectors for the TMS370C8

```
.global _money

.sect ".trapvecs"

.if .TMS370C8

.word 0                         ; TRAP 15
.word 0                         ; TRAP 14
.word 0                         ; TRAP 13
.word _money                    ; TRAP 12
.word 0                         ; TRAP 11
.word 0                         ; TRAP 10
.word 0                         ; TRAP 9
.word 0                         ; TRAP 8
.word 0                         ; TRAP 7
.word 0                         ; TRAP 6
.word 0                         ; TRAP 5
.word 0                         ; TRAP 4
.word 0                         ; TRAP 3
.word 0                         ; TRAP 2
.word 0                         ; TRAP 1
.word 0                         ; TRAP 0

.else

.word 0                         ; TRAP 15
.word 0                         ; TRAP 14
.word 0                         ; TRAP 13
.word _money                    ; TRAP 12
.word 0                         ; TRAP 11
.word 0                         ; TRAP 10
.word 0                         ; TRAP 9
.word 0                         ; TRAP 8
.word 0                         ; TRAP 7
.word 0                         ; TRAP 6
.word 0                         ; TRAP 5
.word 0                         ; TRAP 4
.word 0                         ; TRAP 3
.word 0                         ; TRAP 2
.word 0                         ; TRAP 1
.word 0                         ; TRAP 0

.endif
```

4.6.8 The REENTRANT Pragma

The REENTRANT pragma provides an alternate means of declaring a function to be reentrant:

```
#pragma REENTRANT (fact)
int fact (int n)
{
      if (n==1) return 1;
      return (n * fact(n–1));
}
```

The above definition of the factorial function has the same effect as using the reentrant keyword.

4.6.9 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 may have C functions that are called by hand-coded assembly instead of main. The FUNC_EXT_CALLED pragma allows you to instruct the compiler to keep these C functions or any other functions that these C functions call. These functions act like main; they function as entry points into C.

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

#pragma FUNC_EXT_CALLED (func)

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

When you use program-level optimization, you may need to use the FUNC_EXT_CALLED pragma with certain options. Refer to page 2-28 for information about this situation.

4.7 The asm Statement

The TMS370/C8 8-bit C compiler allows you to embed ’370/C8 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 command is provided so that you can access hardware features, which, by definition, C is unable to access. The asm statement is syntactically like a call to a function called asm with one string constant argument:

asm(“*assembler text*”);

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

```
asm(“STR: .byte \”abc\””);
```

The inserted code must be a legal assembly language statement. Like all assembly language statements, the line of code inside the quotes *must* begin with a label, a blank, a tab, or a comment (asterisk or semicolon). The compiler does no checking on the string; if there is an error, it will be detected by the assembler. For more information, refer to the *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User’s Guide*.

Note: Avoid Disrupting the C Environment With asm Statements

Be careful not to disrupt the C environment with asm statements. The compiler does not check the inserted instructions. Inserting jumps and labels into C code can cause unpredictable results in variables manipulated in or around the inserted code. Directives that change sections or otherwise affect the assembly environment can also be troublesome.

Be especially careful when you use the optimizer with asm statements. Although the optimizer cannot remove asm statements, it can significantly rearrange the code order near them, possibly causing undesired results.

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

Note: Compatibility With New Addressing-Mode Syntax

Code compiled with the old addressing-mode syntax that contains asm statements must be updated before it can be used with the new addressing-mode syntax.

4.8 Initializing Static and Global Variables

The ANSI C standard specifies that global (extern) and static variables, without explicit initializations, must be preinitialized to 0 (zero) (before the program begins running). This task is typically performed when the program is loaded. Because the loading process is heavily dependent on the specific environment of the target application system, the compiler itself makes no provision for preinitializing variables; 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 and the .reg section:

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

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

If your application is to be burned into ROM, you should explicitly initialize variables that require initialization. The method demonstrated above initializes .bss and .reg to zero only at load time, not at system reset or power-up.

For more information about linker command files and the SECTIONS directive, see the *Linker Description* chapter in the *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide.*

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

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

However, there are subtle changes in the language that may affect existing code. Appendix C of *The C Programming Language* (second edition) summarizes the differences between ANSI C and the first edition's C standard (hereafter referred to as K&R C).

To simplify the process of compiling existing C programs with the '370/C8 8-bit 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:

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

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

- ANSI prohibits 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. You can use –pe, which converts code-E errors to warnings, 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 separate definitions, 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 int a is defined twice.

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

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

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

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

- ANSI specifies that bit fields must be of type int or unsigned. Bit fields are limited to eight bits. With –pk, bit fields can be legally declared with any integral type. For example:

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

  Note that the '370/C8 8-bit C compiler operates on bit fields declared as ints as unsigned 8-bit ints. Signed int bit field declarations are prohibited.

- 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 */
```

4.10 Compiler Limits

Due to the variety of host systems supported by the ’370/C8 8-bit 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 are detected by the parser. When the parser detects such a condition, it 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. Simplify the program to avoid exceeding a compiler limit.

Many compiler tables have no absolute limits but rather are limited only by the amount of memory available in the host system. Table 4–3 specifies the limits that are absolute. All of the absolute limits equal or exceed those required by the ANSI C standard.

Table 4–3. Absolute Compiler Limits

Description	Limits
Filename length	512 characters
Source line length	16K characters†
Length of strings built from # or ##	512 characters‡
Inline assembly string length	132 characters
Macros predefined with –d	64
Macro parameters	32 parameters
Macro nesting level	100 levels§
#include search paths	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 parameters
Array, function, or pointer derivations on a type	12 derivations
Aggregate initialization nesting	32 levels
Local initializers	150 levels (approximately)
Nesting of if statements, switch statements, and loops	32 levels

† After splicing of \ lines. This limit also applies to any single macro definition or invocation.
‡ Before concatenation. All other character strings are unrestricted.
§ Includes argument substitutions.
¶ Includes –i and C_DIR directories.

Chapter 5

Runtime Environment

This chapter describes the TMS370/C8 8-bit C runtime environment. To ensure successful execution of C programs, it is critical that all runtime code maintain this environment. It is also important to follow the guidelines described in this chapter if you write assembly language functions that interface to C code.

Topic **Page**

5.1 Memory Model

The C compiler treats memory as a single linear block of memory that is partitioned into sections. Each block of code or data that a C program generates is placed in its own contiguous space in memory. The compiler assumes the full 16-bit address space is available in target memory.

Note: The Linker Defines the Memory Map

The linker, not the compiler, defines the memory map and allocates code and data into target memory. (The only exception to this is that the compiler assumes the .reg section is allocated within the TMS370/C8 internal register file.) The compiler assumes nothing about the types of memory that are available or about any locations that are not available (holes). The compiler produces relocatable code, which 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.

5.1.1 Sections

The compiler produces relocatable blocks of code and data called *sections*, which can be allocated into memory in a variety of ways to conform to a variety of system configurations. For more information about COFF sections, see the *Introduction to Common Object File Format* chapter in the *TMS370 and TMS370C8 8-Bit Microcontroller Family 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 three initialized sections: .text, .cinit, and .const.
  - The **.text section** is an initialized section that contains all of the executable code.
  - 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, switch tables, and data defined with the C qualifier *const* (provided the constant is not also defined as *volatile* or *near*).

- **Uninitialized sections** reserve space in memory (usually in RAM). A program can use this space at runtime for creating and storing variables. The compiler creates five uninitialized sections: .bss, .heap, .hstack, .sstack, and .reg.
  - The **.bss section** is an uninitialized section that reserves space for far 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 .bss. The .bss section is also used for storing far automatic storage class variables which might become overlayed.
  - The **.reg section** is an uninitialized section similar to the .bss section. It reserves space for near global and static variables. However, it is allocated into the TMS370/C8's register file; therefore, any object stored in this section can be accessed very quickly.
  - The **.heap section** is an uninitialized section that reserves space for dynamic memory allocation. The reserved space is used by the malloc, calloc, and realloc functions. If none of these functions are used, the size of the section remains 0.
  - The **.hstack section** is an uninitialized section that reserves space for the TMS370/C8's hardware stack area. The hardware stack is used for saving return addresses during function calls and passing arguments when registers are unavailable. The TMS370/C8 status register is saved here when an interrupt occurs.
  - The **.sstack section** is an uninitialized section that reserves space for the TMS370/C8's software stack area. The software stack is used for storing automatic storage class variables in reentrant functions. The size of the software stack is 0 (zero) by default.

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

The linker takes the individual sections from different modules and combines sections that have the same name. The resulting eleven output sections and the appropriate placement in memory for each section are listed in Table 5–1. You can place these output sections anywhere in the address space, as needed to meet system requirements.

Table 5–1. Summary of Sections and Memory Placement

Section	Type of Memory	Section	Type of Memory
.text	ROM or RAM	.hstack	RAM
.cinit	ROM or RAM	.sstack	RAM
.const	ROM or RAM	.heap	RAM
.data	ROM or RAM	.reg	internal register file
.bss	RAM	.regovy	internal register file
.bssovy	RAM		

† The .hstack section must be linked to the internal register file for the TMS370 and to the first 1K bytes of RAM for the TMS370C8.

For more information about allocating sections into memory, see the TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide.

5.1.2 Subsections

Sections can be broken down into subsections, allowing you to allocate memory at a deeper level. A subsection can be defined as either initialized or unutilized.

There are two types of subsections:

- **Initialized subsections** are defined by a .sect directive.

  .sect **"***section name* **:** *subsection name***"**[**,***address*]

  For example, the compiler generates code for a function into a .text subsection:

  ```
  .sect ".text:_myfunc"
  ```

  which causes the code generated for a function called myfunc to be placed in a section called .text:_myfunc.

The compiler also generates a .clink directive for every .text subsection. This directive tells the linker that if the subsection is not referenced, it can be left out of the linked executable file, even though another subsection within the same file might be referenced. This allows you to develop source modules with several functions in them without having to worry about the object associated with a function being included in the resulting linked object if the function is never called.

- **Uninitialized subsections** are used by the compiler to hold local variables that reside in memory. These subsections are defined by a .usect directive:

  symbol **.usect** **"***section name* **:** *subsection name***"**, *size in bytes*

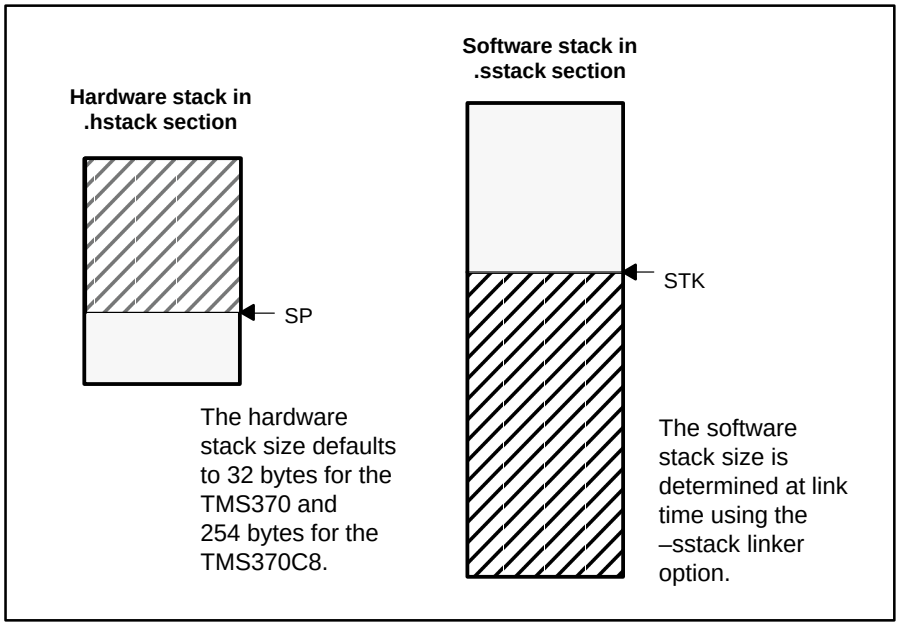
  The linker uses call graph information that is generated by the compiler to determine whether two functions are ever active at the same time. In a call graph, if one function is not a descendant of another function, the two functions are never active simultaneously.

  This means that the linker can allocate an uninitialized subsection associated with one function into the same space as the uninitialized subsection for another function if the two functions are never active at the same time.

5.1.3 C System Stacks

The TMS370/C8 8-bit C compiler uses a hardware stack and a software stack. Figure 5–1 illustrates these stacks in memory.

Figure 5–1. C System Stacks



5.1.3.1 Hardware Stack

The hardware stack is used by the TMS370/C8 to:

- Save the return address for call instructions and interrupts
- Save the status register on interrupts
- Pass arguments when the number of available argument registers is exhausted
- Pass variable length argument lists to functions that have variable numbers of arguments

The compiler uses the stack pointer (SP) to manage the hardware stack.

The linker allocates memory for the hardware stack in an uninitialized, named section called .hstack. The default size is 32 bytes for the TMS370 and 254 bytes for the TMS370C8; the maximum size is 254 bytes for both. You can change the amount of memory reserved for the hardware stack by invoking the linker with the –hstack option. Specify a constant directly after the option. For example:

```
lnk370 –hstack 050h  /*defines an 80 byte hardware stack*/
```

For more information, see the *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide*.

At system initialization, SP is set to a designated address for the top of the hardware stack. This address is the first location in the .hstack section; because the position of the stack depends on where the .hstack section is allocated, the actual position of the hardware stack is determined at link time. Note, however, that the entire .hstack section must be allocated within the register file (addresses 0–255) for the TMS370, and within the first 1K bytes of RAM for the TMS370C8.

The linker determines where the .hstack section is allocated. If you want .hstack to be the last section allocated in the register file, move the .hstack section to the last output section listed in the SECTIONS directive in the linker command file. The _ _HSTACK_SIZE constant is created by the linker and represents the size of the .hstack section.

Note: Hardware Stack Overflow

The compiler provides no means to check for hardware stack overflow during compilation or at runtime. A stack overflow disrupts the runtime environment, causing your program to fail unpredictably. Be sure to allow enough space for the stack to grow; use the –hstack linker option. The TMS370C8 provides a hardware stack overflow reset ability. For more information, see the *TMS370C8 CPU and Memory Organization* chapter of the *TMS370C8 Central Processing Unit, System, and Instruction Set Reference Guide.*

5.1.3.2 Changing the Hardware Stack Location (TMS370C8 Only)

For the TMS370C8, the hardware stack can reside anywhere within the first 1K bytes of RAM (the first four pages of RAM). The sample linker command file included with the compiler places the hardware stack in the first page of RAM (register file), as with the TMS370. If you want to place the stack in another page, you must allocate the .hstack section into the desired page. For example, to place the hardware stack into the second page of RAM, use the linker command file in Example 5–1.

Example 5–1. Changing the Hardware Stack Location

```
MEMORY
{
      RFILE     : org = 0eh        len = 00F2h
      HS_MEM    : org = 280h       len = 0080h
      D_MEM     : org = 2000h      len = 4000h
      P_MEM     : org = 6000h      len = 9FECh
      TVEC_MEM  : org = 7fc0h      len = 0020h
      IVEC_MEM  : org = 0FFECh     len = 0014h
}
SECTIONS
{
      .reg      : {} > RFILE
      .regovy   : {} > RFILE
      .hstack   : {} > HS_MEM
      .bss      : {} > D_MEM
      .bssovy   : {} > D_MEM
      .heap     : {} > D_MEM
      .sstack   : {} > D_MEM
      .trapvecs : {} > TVEC_MEM
      .intvecs  : {} > IVEC_MEM
      .text     : {} > P_MEM
      .const    : {} > P_MEM
      .cinit    : {} > P_MEM
}
```

The boot routine in the runtime-support library uses the location of the .hstack section to set the stack page bits (STK PAGE 1, STK PAGE 0) in system control register 1 (SCR1). Therefore, you must allocate the .hstack section within the first four pages of RAM. For more information about hardware stack pages, refer to the *TMS370C8 Central Processing Unit, System, and Instruction Set Reference Guide.*

Note: Changing the Hardware Stack for TMS370C8

If overflow detection is enabled, do not allocate the hardware stack at the beginning of a page. The compiler decrements the stack pointer when setting up the hardware stack; if the stack begins at the beginning of a page, this causes an overflow condition.

5.1.3.3 Software Stack

The software stack is used by the TMS370/C8 to provide dynamic storage for local variables in reentrant functions. The .sstack section is typically allocated to RAM, along with the .bss section or the .heap section. The compiler uses a global relocatable register (STK) that is defined in boot.c to manage the software stack.

The compiler dynamically allocates memory for local variables in a reentrant function by subtracting the space needed for reentrant local variables from STK on entry into a reentrant function and adding that value back into STK prior to exiting a reentrant function. You must reserve enough space in the .sstack section to handle the local variables of all of the reentrant functions that are active during the execution of your program.

At system initialization, STK points to the end of the .sstack section. There is no space allocated to the software stack by default. You can define the size of the software stack at link time using the –sstack option within a linker command file or on the linker command line. For example:

```
lnk370 –sstack 100h /*defines a 256 byte software stack*/
```

The linker creates the constant _ _SSTACK_SIZE that represents the size of the .sstack section.

Note: Software Stack Overflow

The compiler provides no means to check for software stack overflow during compilation or at runtime. A stack overflow disrupts the runtime environment, causing your program to fail unpredictably. Be sure to allow enough space for the stack to grow; use the –sstack linker option. For more information, see Chapter 2 of the *TMS370C8 Central Processing Unit, System, and Instruction Set Reference Guide.*

5.1.4 Dynamic Memory Allocation

Dynamic allocation is not a standard part of the C language; it is provided by standard runtime-support functions. The compiler's runtime-support library 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 these functions to allocate memory from the heap.

The linker defines the size of the .heap section and sets a constant, _HEAP_SIZE, equal to the size of the section. The default size of the .heap section is 512 bytes. You can change this size by invoking the linker with the –heap option. Specify the memory size as a constant directly after the option. For example:

```
lnk370 –heap 0400h    /* defines a 1K heap */
```

For more information, see the *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide*.

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 that are 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 initialize variables in 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. Your 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 about autoinitialization, see the *Linker Description* chapter in the *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide*.

5.2 Data Representation and Storage

This section explains how various data types are represented and stored in memory and how to use character string constants.

5.2.1 Data Type Storage

No packing is performed on any data item except for bit fields, which are packed into bytes. Bit fields are allocated from the MSB to the LSB in the order in which they are declared.

Objects do not have any type of alignment requirements; any object can be stored on any byte boundary. Objects that are members of structures or arrays are stored the same way as individual objects.

Table 5–2 lists register and memory storage in the ’370/C8 for various data types:

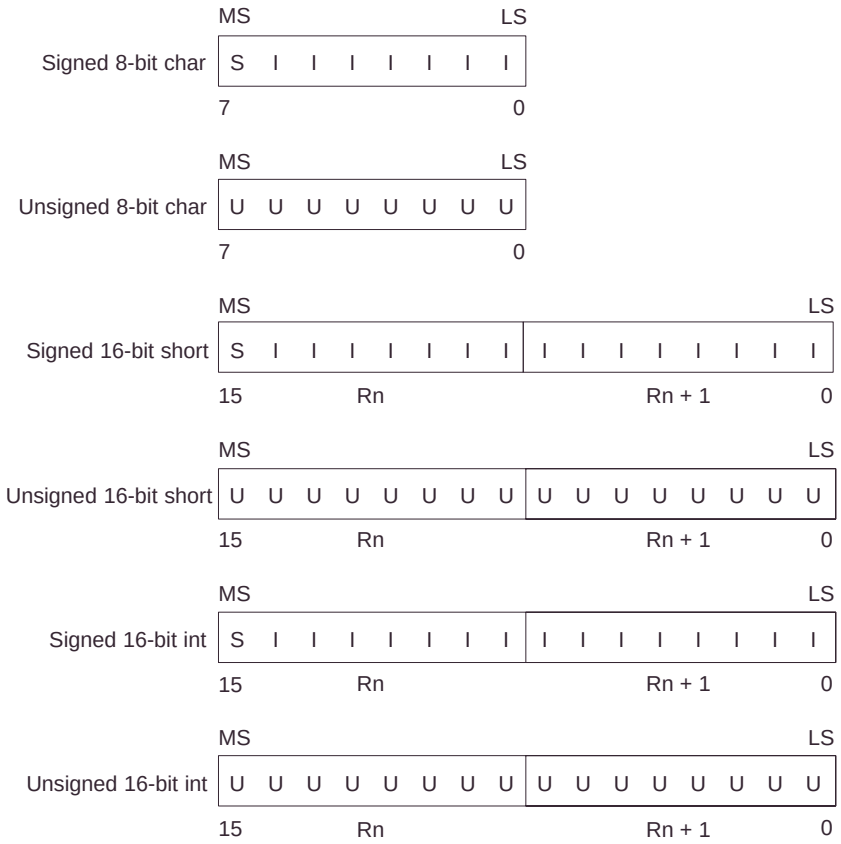
Table 5–2. Data Representation in Registers and Memory

Data Type	Register Storage	Memory Storage
char, signed char	bits 0–7 of register†	1 byte
unsigned char	bits 0–7 of register	1 byte
short, signed short	bits 0–15 of register†	2 bytes
unsigned short	bits 0–15 of register	2 bytes
int, signed int	bits 0–15 of register	2 bytes
unsigned int	bits 0–15 of register	2 bytes
enum	bits 0–15 of register	2 bytes
long, signed long	bits 0–31 of register	4 bytes
unsigned long	bits 0–31 of register	4 bytes
float	bits 0–31 of register	4 bytes
double	bits 0–31 of register	8 bytes
long double	bits 0–31 of register	8 bytes
struct	members stored as their individual types require	members stored as their individual types require
array	members stored as their individual types require	members stored and aligned as their individual types require

† Negative values are sign-extended to bit 31.

Char and unsigned char objects are stored in memory as a single byte, and they are loaded to and stored from bits 0–7 of a register (see Figure 5–2). Objects defined as short, unsigned short, int, or unsigned int are stored in memory as two bytes, and they are loaded to and stored in two registers (see Figure 5–2).

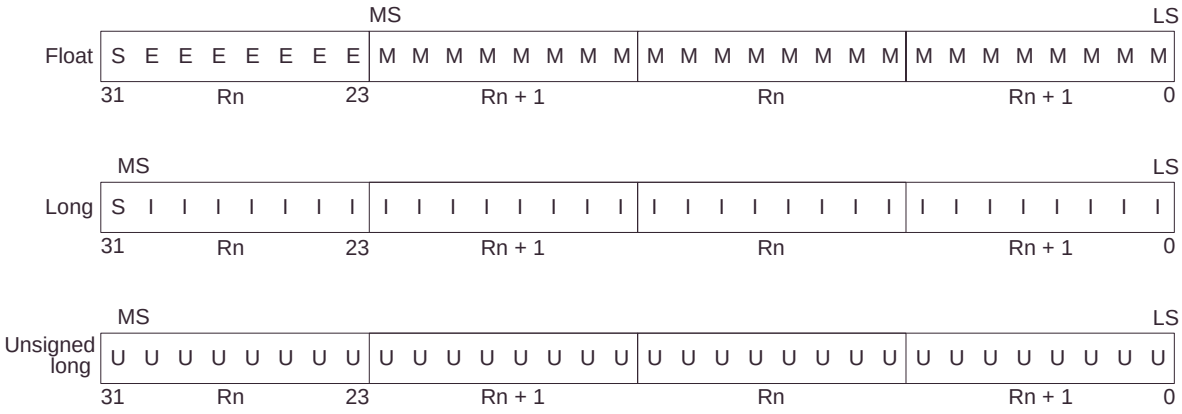
Figure 5–2. char, short, and int Data in ’370/C8 Registers



Legend: S = sign
I = signed integer
U = unsigned integer

Three data types—long, unsigned long, and float—are stored in memory as 32-bit objects. Objects of these types are loaded to and stored in four registers, as shown in Figure 5–3.

Figure 5–3. 32-Bit Data in '370/C8 Registers



Legend: S = sign E = exponent
M = mantissa I = signed integer
U = unsigned integer

5.2.2 Bit Fields

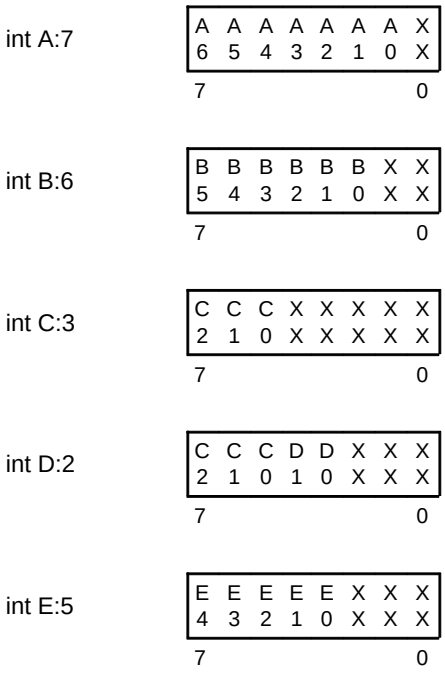
Bit fields are the only objects that are packed within a byte; that is, two bit fields can be stored in the same byte. Bit fields can range in size from one to eight bits, but they will never span a 1-byte boundary.

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

```
struct{
unsigned int A:7
unsigned int B:6
unsigned int C:3
unsigned int D:2
unsigned int E:5
}x;
```

A0 represents the least significant bit of the field A; A1 represents the next least significant bit, etc.

Figure 5–4. Bit-Field Packing Format



5.2.3 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 treated as an initialized array; each character is a separate initializer. For more information about autoinitialization, see Section 5.9, page 5-36.

- 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 .text section by using the .string assembler directive along with a unique label that points to the string; the terminating 0 byte is included. The following example defines the string abc, along with the terminating byte; the label SL5 points to the string:

  ```
        .text
        .align   4
  SL5:  .string  ″abc″, 0
  ```

  String labels have the form **SL***n*, where *n* is a number assigned by the compiler. Numbering begins at 0 and increases by 1 for each defined string. The label **SL***n* represents the address of the string constant. The compiler uses this label to reference the string in the expression.

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

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

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

5.3 Register Conventions

Strict conventions associate specific registers with specific operations in the C environment. If you plan to interface assembly language routines to a C program, you must follow these register conventions.

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.

Save-on-call and static register blocks are two types of register variable registers. The distinction between these two types of register-variable registers is how they are preserved across calls.

- **Save-on-call registers** are registers R0–R13. The called function does not preserve the values in these registers; therefore, the calling function must save them if their values need to be preserved.
- **Static register blocks** are allocated from registers R14–R255. Because the runtime model is non-reentrant, the compiler assumes by default that these registers are never modified except in the function that references these registers.

When a function uses save-on-call registers, at every call, the function must save and restore any registers whose contents are required both before and after the call. Table 5–3 lists the registers, their types, and how they are to be used.

Table 5–3. Register Use Conventions

Registers	Type	Usage
R0 (A), R1 (B)	expression register	The compiler uses the A and B registers as general-purpose expression registers.
R2–R13	argument register	The compiler uses these registers for passing arguments from caller to called functions, and they are also used as general-purpose expression registers.

Registers R14–R255 are available for other uses. Remember that the .reg section must be linked into these remaining registers. Any registers remaining after the .reg and .hstack sections are allocated can be used for application-specific purposes.

5.3.1 Expression Registers

The compiler uses registers A, B, and R2–R13 for evaluating expressions and storing temporary results. The compiler keeps track of the current contents of each register and attempts to allocate registers for expressions in a way that preserves useful contents in the registers whenever possible. This allows the compiler to reuse register data and take advantage of the TMS370/C8's efficient register-addressing modes. This allocation scheme also avoids unnecessary accessing of variables and constants.

When a function is called, the compiler forgets the contents of the expression registers. Any value that is active across a call is not allocated to a save-on-call register.

5.3.2 Return Values

When values are returned from functions, the compiler places these values in particular registers as shown in Table 5–4.

Table 5–4. Return Values

If the return value is ...	it is placed in register...
char	R2
int or short	R2:R3
floating point	R2:R3:R4:R5
long	R2:R3:R4:R5

5.3.3 Register Variables

Register variables are local variables that you can define (or the compiler can generate) to reside in registers rather than in memory. Storing local variables in registers allows significantly faster access, which improves the efficiency of compiled code.

- **When you do not use the optimizer** (–o flag), you can allocate register variables into static register blocks with the register keyword.The number of registers allocated for a register variable depends on the number of bytes used to represent its type:
  - A floating-point register variable requires four registers.
  - A long integer register variable requires four registers.
  - An int or short register variable requires two registers.
  - A char register variable requires one register.
- **When you use the optimizer**, the compiler ignores the register keyword and uses a cost-analysis algorithm to allocate variables and temporaries.

5.4 Overlaying Automatic Variables

The TMS370/C8 can overlay register blocks and uninitialized memory sections that contain automatic data, which saves data space in the .bss section and the register file. This capability is enabled by default unless it is explicitly disabled by the –mc shell option. The compiler generates information into the COFF file that the linker uses to construct a call graph. From the call graph, the linker determines which functions are never active at the same time. Then the linker allocates the automatic data space for these functions into the same location.

5.4.1 Overlaying Local Register Blocks

When a function requires the use of static register block registers to preserve values across a function call, the function defines an overlay register block subsection. This section has .reg as its base name and the current function's name as its subsection name. For example, in the function foobar, the required static register block registers are defined as follows:

```
.reg  ".reg:_foobar", 10
```

which creates a section called .reg:_foobar that contains 10 bytes of uninitialized data to be allocated into the TMS370/C8 register file.

A second function, foobee, also uses a static register block of size 10:

```
.reg  ".reg:_foobee", 10
```

The compiler generates call graph directives (when –mc is not used on the shell command line) that provide information to the linker about which functions call which other functions. With this information, the linker can determine whether two functions are active at the same time. If the two functions are never active at the same time, the .reg subsections that are associated with both functions are allocated to the same location within the register file.

5.4.2 Overlaying Local Memory Blocks

The compiler and linker apply the same technique to blocks of uninitialized sections that are associated with two functions that are known never to be active at the same time. Suppose that foobar and foobee both contain local arrays that are allocated to their respective .bss sections. The compiler generates:

```
$B_foobar .usect ".bss:_foobar", 20
$B_foobee .usect ".bss:_foobee", 20
```

Using the same call graph information, the linker determines that foobee and foobar are never active at the same time. This allows the linker to allocate both .bss subsections, .bss:_foobar and .bss:_foobee to the same location in memory.

5.5 Function-Calling Conventions

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

5.5.1 How a Function Makes a Call

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

1) When a C function is called from assembly language, registers R2 through R13 are available for passing arguments. Arguments are packed in registers R2 through R13 in the size and order in which they appear in the function call.

2) If there are fewer argument registers than are required to hold all of the arguments, then extra arguments are placed on the hardware stack in the reverse order in which they appear in the function call. See Figure 5–5.

3) If the function being called has a variable number of arguments, then the last argument passed before the list of variable arguments and all arguments in the variable argument list are placed on the hardware stack in the reverse order in which they appear in the function call.

4) The caller calls the function.

5) When the called function has completed execution and debug mode is not in effect, the caller must remove any arguments that are placed on the hardware stack by subtracting *n* from the value of the hardware stack (where *n* is the total size of all the arguments that were placed on the hardware stack).

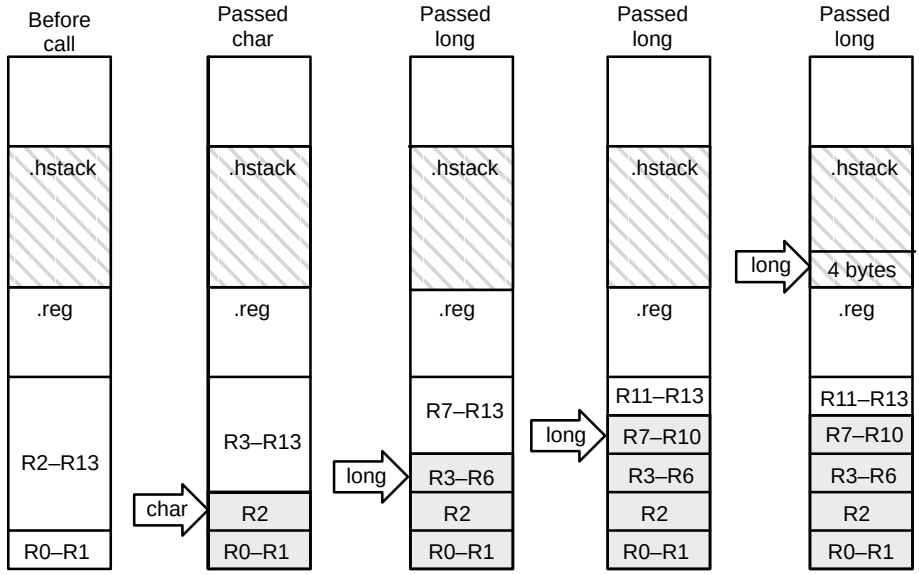
Note: Stack Use Is Different in Debug Mode

Management of the hardware stack space is different in debug mode than it is in normal mode. In normal mode, special stack management allows the compiler to take advantage of the efficiency of PUSH instructions to store arguments on the hardware stack. The dynamic movement of the SP register during a function makes it difficult for the debugger to determine the value of items that may be stored on the stack.

For this reason, debug mode employs an SP-relative addressing mode to place arguments on the stack. This allows the SP to remain fixed throughout the function and allows the debugger to track values more effectively.

Figure 5–5 shows how the stack is used during a function call. In this example, four arguments are passed to the function: one type char and three type long. Char is passed in registers R0– R2. The first long is passed in registers R3–R6. The next long is passed in registers R7–R10. Because there is not enough space in registers R11–R13 for the last long, which requires four registers, this argument is placed on the hardware stack, leaving registers R11–R13 empty.

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



5.5.2 How a Called Function Responds

A called function must perform the following tasks:

1) Allocate space within a register block or save-on-call registers for near local variables

2) Allocate space within an uninitialized section (.bss, for example) for far local variables

3) Execute the code for the function

4) Place the return value, if there is one, in the appropriate registers. Eight-bit return values are placed in register R2, 16-bit return values are placed in R2:R3, and 32-bit return values are placed in R2:R3:R4:R5.

5.5.3 Requirements of a Called Reentrant Function

Functions that are defined with the reentrant keyword or pragma have a different set of requirements than nonreentrant functions (the default). The values stored in local variables and parameters must be kept separate in instances of the reentrant function that may be active simultaneously. To keep these values separate, a reentrant function must perform the following tasks when called:

1) Allocate space on the software stack for local variables as needed

2) Move an argument that is passed in a register onto the software stack for the time the reentrant function is active if it is determined that the argument is live across a function call

3) Execute the code for the function

4) Place the return value, if there is one, in the appropriate registers. Eight-bit return values are placed in register R2, 16-bit return values are placed in R2:R3, and 32-bit return values are placed in R2:R3:R4:R5

5) Deallocate all space on the hardware stack and software stack before returning to the caller

5.5.4 Returning Structures From Functions

A special convention applies to functions that return structures. The caller allocates space for the structure and then passes the address of the return space to the called function as an extra parameter. The called function copies the structure directly to the memory block pointed to by that extra parameter. For example, in the statement:

s = f(x)

s is a structure, x is an 8-bit unsigned character, and f is a function that returns a structure. The caller can actually make the call as s = f(&s,x). The called function will have the address of the return area in the first argument register pair (R2:R3), so it can copy the return structure directly into s to perform the assignment.

If the caller does not use the return value, then an address of 0 can be passed. This directs the called function not to copy the return structure.

You must be careful to properly declare functions that return structures both at the point where they are called (so that the extra parameter is passed) and at the point where they are defined (so that the function copies the result).

5.5.5 Setting Up the Local Environment

If local storage is needed, called C functions allocate space for local variables and local temporary storage by allocating a subsection for the local data with the name R$_*function name*. For example, if a function such as swap(int x, int y) needs three additional bytes of local storage, then the directive below is used to allocate it.

```
.reg   ".reg:_swap", R$_swap, 3
```

5.5.6 Accessing Near Local Variables

A function accesses near local variables directly as register operands. This addressing mode can be used because the local variable block is allocated in the .reg section, and the .reg section is mapped entirely within the TMS370/C8 register file.

Local variables are accessed by adding the appropriate offset to the name of the local variable block. This name is R$_*function name*. For example, the following function's local variables are accessed as indicated here:

```
f()
{
      char  c;     /* 1 byte */
      long  l;     /* 4 bytes */
      short s;     /* 2 bytes */
      .
      .
      .
}
```

c can be referenced as location	R$_f
l can be referenced as locations	R$_f+1, R$_f+2, R$_f+3 and R$_f+4
s can be referenced as locations	R$_f+5 and R$__f+6

Because all local data blocks are allocated in the .reg section and the .reg section must fit entirely in the TMS370/C8 register file (256 bytes), there is a limit to the total number of bytes that can be allocated for these purposes. Also, remember that the total size of the .reg section includes data defined in all the linked .obj files, including the C runtime-support routines. Data space is preserved by the automatic overlay capability and the use of subsections.

5.6 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 as described in this subsection 5.6.1. This is the most versatile method.
- Use inline assembly language embedded directly in the C source (refer to subsection 2.5.4.1, page 2-36).
- Modify the assembly language code that the compiler produces (refer to subsection 5.6.4, page 5-29).

5.6.1 Using Assembly Language Modules With C

If you follow the register conventions defined in Section 5.3 and the calling conventions defined in Section 5.5, interfacing C with assembly language functions is straightforward. C code can access variables and call functions that are defined in assembly language, and assembly code can access C variables and call C functions.

Follow these guidelines to interface assembly language and C:

- You must preserve any dedicated registers that are modified by a function. Dedicated registers include:
  - SP
  - Any registers in the range R14–R255 that are allocated to the .reg or .hstack sections

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

  Any register that is not dedicated can be used freely without first being saved.
- Interrupt routines must save *all* the registers they use. For more information about interrupt handling, refer to Section 5.7, page 5-30.
- Functions must return values correctly according to their C declarations. Nonstructure return locations are described in subsection 5.3.3, on page 5-18; structure return locations are described in subsection 5.5.4, on page 5-22.

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

- The compiler adds 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 a leading 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. If the object is defined in the .reg section, the directive .globreg is used instead.

  Likewise, to access a C function or object from assembly language, declare the C object with .global (or .globreg if the object is allocated in the .reg section). This creates an undefined external reference that the linker will resolve.

Example 5–2 illustrates a C function called main, which calls an assembly language function called asmfunc. The asmfunc function takes its single argument, adds it to the C global variable called gvar, and returns the result.

Example 5–2.An Assembly Language Function

(a) C program

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

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

(b) Assembly language program

```
    ac370 main2.c main2.if
    .sfunc     _main, ".reg:_main", 0, ".bss:_main", 0
    .sect   ".text:_main"
    .clink
    .global    _main

_main:
         CALL      _asmfunc
         .calls    _main, _asmfunc
         RTS

    .globreg   _gvar
    .reg       _gvar,1
;*****************************************************
;* UNDEFINED EXTERNAL REFERENCES                     *
;*****************************************************
           .global               _asmfunc
```

In the C program in Example 5–2(a), the extern declaration of asmfunc is required, because the return type is char. Like C functions, assembly functions need to be declared only if they return nonintegers.

Note the underscores on all the C symbol names in the assembly language code in Example 5–2(b).

5.6.2 How to Define Variables in Assembly Language

It is sometimes useful for a C program to access variables that are 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 far and access it normally.

Example 5–3 shows an example of accessing a variable defined in .bss.

Example 5–3.Accessing a Variable Defined in .bss From C

(a) Assembly language program

```
; Note the use of underscores in the following lines
   .bss     _var,1     ; Define the variable
   .global  _var       ; Declare it as external
```

(b) C program

```
extern far char var;    /* External variable */
var = 1;             /* Assign a value to the variable */
```

Accessing uninitialized variables from the .reg section is also straightforward:

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

Example 5–4 shows an example of accessing a variable defined in .reg.

Example 5–4.Accessing a Variable Defined in .reg From C

(a) Assembly language program

```
; Note the use of underscores in the following lines
   .reg     _var,1     ; Define the variable
   .globreg  _var      ; Declare it as external
```

(b) C program

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

You can also access a variable that is not defined in the .bss or .reg sections. For example, you might want to have a look-up table defined in assembly language that you don't want to put in RAM. The following procedure outlines the access process:

1) Define the variable in a named section other than .reg or .bss.

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 far and access it normally.

Example 5–5 shows an example of accessing a variable that is not defined in .bss or .reg.

Example 5–5.Accessing a Variable That Is Not Defined in .bss or .reg From C

(a) Assembly language program

```
    .global   _mytab    ; Declare variable as external
    .sect  ”rom_tab”    ; Make a separate section
_mytab                  ; The table starts here
    .word  14
    .word  28
    .word  42
```

(b) C program

```
    extern far int mytab[];    /* This is the object  */
    int l;
    l = mytab[2];              /* Access my tab like a */
                               /* normal array         */
```

5.6.3 Using 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 will place sequential lines of assembly language into the compiler output with no intervening code. For more information about the asm statement, see Section 4.7, page 4-19.

Note: Using the asm Statement

1) The asm statement is provided so that you can access features of the hardware that would otherwise be 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.

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

3) Do not change the value of a C variable when using an asm statement (you can safely *read* the current value of any variable).

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

The asm statement is also useful for inserting comments in the compiler output; simply start the assembly code string with a semicolon (;), as shown here:

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

5.6.4 Modifying Compiler Output

You can inspect and change the compiler's assembly language output by compiling the source and then editing the assembly output file before assembling it. The C interlist utility can help you inspect compiler output. (For information on the interlist utility, see Section 2.6, page 2-40.) Also refer to the note in subsection 2.5.4.1, page 2-36, about disrupting the C environment, which applies to modification of compiler output. Changes can be overwritten if the C source file is recompiled.

5.7 Interrupt Handling

If 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, you must handle any required enabling or masking of interrupts. Such operations have no effect on the C environment and can easily be incorporated with asm statements.

5.7.1 Using C Interrupt Routines

Interrupts can be handled directly with C functions by using either the interrupt keyword or interrupt pragma or a special naming convention. See subsection 4.5.2, page 4-10, for information on the interrupt keyword or subsection 4.6.4, page 4-14, for information on the interrupt pragma. C interrupt functions have names with the following format:

`c_int`*nn*

where *nn* is a two-digit number between 00 and 99 (for example, a valid interrupt routine name is c_int01). By following this convention, you ensure that the compiler uses the special register preservation requirements discussed in subsection 4.5.2.

The name c_int00 is reserved for the system reset interrupt. This special interrupt routine initializes the system and calls the function main; c_int00 does not save any registers, because it has no caller.

If a C interrupt routine does not call any other functions; only those registers that are actually used in the interrupt handler are saved and restored. However, if a C interrupt routine *does* call other functions, these functions may modify unknown registers that are not used in the interrupt handler itself. For this reason, the routine saves all of the expression registers and all of the save-on-call register variable registers if any other functions are called.

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

Interrupt routines that have local data are *not* reentrant routines by default, because the local data is allocated statically in the .reg section. You can declare the interrupt routine to be reentrant. Local data would then be placed on the software stack. If you do not define the interrupt routine as reentrant, either do not define local data, or do not reenable the interrupt that vectors to this routine when the routine is entered.

5.7.2 Using Assembly Language Interrupt Routines

Interrupts can be handled with assembly language code as long as the register conventions are followed. Like all assembly functions, interrupt routines can use the stack, access global C variables, and call C functions normally. When calling C functions, be sure that all expression and save-on-call register-variable registers are preserved before the call, because the C function can modify any of them. Register-variable allocated to a static register block need not be saved, because they are referenced only in called C function.

5.7.3 How to Map Interrupt Routines to Interrupt Vectors

The compiler tools include a file called intvecs.asm. This file contains assembly language directives that can be used to set up the TMS370/C8's interrupt vectors with the addresses of your interrupt routines. Follow these steps to use this file:

1) Update the file intvecs.asm to include your interrupt routines. For each routine:
   a) At the beginning of the file, add a .global directive that names the routine.
   b) Modify the appropriate .word directive by replacing the 0 with the name of your interrupt routine. The comments in the intvecs.asm file indicate which .word directive is associated with which interrupt vector.

Note: Prefixing C Interrupt Routines

Remember, if you are using C interrupt routines, you must prefix the names with an underscore. For example, use _c_int03 instead of c_int03.

2) Assemble and link intvecs.asm with your applications code and with the compiler's link control file (lnk.cmd or lnkc8.cmd). The control file contains a SECTION directive that maps the .intvecs section into the memory locations 0x07FEC–0x07FFF for the TMS370 or 0xFFEC–0xFFFF for the TMS370C8.

For example, if you have written a C interrupt routine called c_int03 for the external interrupt 3 and an assembly language routine called tim1_int for the timer 1 interrupt, then you should modify intvecs.asm as follows:

```
.global  _c_int00
.global  _c_int03
.global  tim1_int

.sect ".intvecs"
.word 0                 ;  a to d interrupt vector
.word 0                 ;  timer 2 interrupt vector
.word 0                 ;  sci tx interrupt vector
.word 0                 ;  sci rx interrupt vector
.word tim1_int          ;  timer 1 interrupt vector
.word 0                 ;  spi interrupt vector
.word _c_int03          ;  external interrupt 3 vector
.word 0                 ;  external interrupt 2 vector
.word 0                 ;  external interrupt 1 vector
.word _c_int00          ;  reset interrupt vector
```

5.8 Expression Analysis

All C expressions are calculated by using registers A, B, and R2–R13, which are designated for evaluating expressions.

Expressions are evaluated according to standard C precedence rules. When a binary operator is analyzed, the order of evaluation of the operands is based on their relative complexity. The compiler tries to evaluate subexpressions in a way that minimizes saving temporary results, which are calculated in registers and saved in memory. This does not apply to operators that specify a particular order of evaluation (such as the comma, &&, and ||), which are always evaluated in the correct order, left to right.

5.8.1 Runtime-Support Arithmetic Routines

The TMS370/C8 instruction set does not support all features of the C language. For some combinations of C arithmetic operators and operand data types, you must call runtime-support functions to perform the operation. For integer arithmetic, these combinations are any variable shift operation, 32-bit integer multiply and divide, 16-bit multiply, or integer division (except for unsigned 8-bit division).

The TMS370/C8 C compiler supports an 8-bit $\times$ 8-bit multiplication, resulting in a 16-bit product. This feature takes advantage of the TMS370/C8's hardware multiply instruction. To use this feature, you must either assign the result of the multiplication to an int or short or cast the result to an int or short. For example, the following two statements use the 8-bit $\times$ 8-bit multiply.

```
int  int1=  char1*char2;
int  int2=  int1 + (int) (char3*char4);
```

Normally, in C, the result of the multiply would be a truncated 8-bit result, which would then be sign-extended. For example, if in the two statements above, char1 had a value of 12 and char2 had a value of 28, int1 would be assigned the value 336 (or 0x0150) using the 8-bit $\times$ 8-bit multiply. This same example would be assigned the value 80 (or 0x0050) using normal C multiplication.

Because the TMS370/C8 does not provide any hardware support for floating-point values, all of the legal C operations on floating-point objects must be simulated by runtime-support functions.

Any registers that are used by these runtime-support routines are saved, except for defined input registers and defined output registers. Input values to these routines are preserved only in certain routines. Refer to Table 5–5 for more information.

The runtime-support math functions are written in assembly language. Source code for them is provided in the source library rts.src. You must use the mk370

library-build utility to create a runtime library for your application, and then use the library at link time by using the –l (lowercase L) option.

The source code has comments that describe the operation and timing of the functions. You can extract, inspect, and modify any of the math functions; be sure you follow the special calling conventions and register saving rules outlined in this section.

Table 5–5 describes the runtime-support math functions, names the files that contain the functions, and summarizes the input and output conventions.

Table 5–5. Input and Output Conventions for Internal Runtime-Support Math Functions

			Inputs		
Function	**Description**	**Source File**	**op1**	**op2**	**Outputs**
ashr_8	8-bit arithmetic shift right	8_ashr.asm	R2	R3	R2
shl_8	8-bit logical shift left	8_shl.asm	R2	R3	R2
shr_8	8-bit logical shift right	8_shr.asm	R2	R3	R2
ashr_16	16-bit arithmetic shift right	16_ashr.asm	R2:R3	R4	R2:R3
sdiv_16	16-bit signed divide	16_sdiv.asm	R2:R3	R4:R5	QUO = R2:R3
shl_16	16-bit logical shift left	16_shl.asm	R2:R3	R4	R2:R3
shr_16	16-bit logical shift right	16_shr.asm	R2:R3	R4	R2:R3
smod_16	16-bit signed modulus	16_smod.asm	R2:R3	R4:R5	R2:R3
smul_16	16-bit signed multiply	16_smul.asm	R2:R3	R4:R5	R2:R3
udiv_16	16-bit unsigned divide	16_udiv.asm	R2:R3	R4:R5	QUO = R2:R3
umod_16	16-bit unsigned modulus	16_umod.asm	R2:R3	R4:R5	R2:R3
umul_16	16-bit unsigned multiply	16_umul.asm	R2:R3	R4:R5	R2:R3
ashr_32	32-bit arithmetic shift right	32_ashr.asm	R2::R5	R6	R2::R5
neg_32	32-bit negate	32_neg.asm	R2::R5	R2::R5	R2::R5
scmp_32	32-bit signed compare	32_scmp.asm	R2::R5	R6::R9	ST is set based on op1 – op2
sdiv_32	32-bit signed divide	32_div.asm	R2::R5	R6::R9	QUO = R2::R5
shl_32	32-bit logical shift left	32_shl.asm	R2::R5	R6	R2::R5
shr_32	32-bit logical shift right	32_shr.asm	R2::R5	R6	R2::R5
smod_32	32-bit signed modulus	32_smod.asm	R2::R5	R6::R9	R2::R5
smul_32	32-bit signed multiply	32_smul.asm	R2::R5	R6::R9	R2::R5
ucmp_32	32-bit unsigned compare	32_ucmp.asm	R2::R5	R6::R9	ST is set based on op1 – op2
udiv_32	32-bit unsigned divide	32_udiv.asm	R2::R5	R6::R9	QUO = R2::R5

Table 5–5. Input and Output Conventions for Internal Runtime-Support Math Functions (Continued)

			Inputs		
Function	**Description**	**Source File**	**op1**	**op2**	**Outputs**
umod_32	32-bit unsigned modulus	32_umod.asm	R2::R5	R6::R9	R2::R5
umul_32	32-bit unsigned multiply	32_umul.asm	R2::R5	R6::R9	R2::R5
fp_add	floating-point add	fp_add.asm	R2::R5	R6::R9	R2::R5
fp_sub	floating-point subtract	fp_add.asm	R2::R5	R6::R9	R2::R5
fp_cmp	floating-point compare	fp_cmp.asm	R2::R5	R6::R9	ST is set based on op1 – op2
fp_div	floating-point divide	fp_div.asm	R2::R5	R6::R9	R2::R5
fp_ftos8	convert float to 8-bit signed char	fp_ftoc.asm	R2::R5	none	R2
fp_ftos16	convert float to 16-bit signed int	fp_ftoi.asm	R2::R5	none	R2:R3
fp_ftos32	convert float to 32-bit signed long	fp_ftol.asm	R2::R5	none	R2:R3
fp_ftou8	convert float to 8-bit unsigned char	fp_ftouc.asm	R2::R5	none	R2
fp_ftou16	convert float to 16-bit unsigned int	fp_ftou.asm	R2::R5	none	R2:R3
fp_ftou32	convert float to 32-bit unsigned long	fp_ftoul.asm	R2::R5	none	R2::R5
fp_dec	floating-point decrement	fp_inc.asm	R2::R5	none	R2::R5
fp_inc	floating-point increment	fp_inc.asm	R2::R5	none	R2::R5
fp_s8tof	convert 8-bit signed char to float	fp_ctof.asm	R2:R3	none	R2::R5
fp_s16tof	convert 16-bit signed int to float	fp_itof.asm	R2::R5	none	R2::R5
fp_s32tof	convert 32-bit signed long to float	fp_ltof.asm	R2::R5	none	R2::R5
fp_mpy	floating-point multiply	fp_mul.asm	R2::R5	R6::R9	R2::R5
fp_neg	floating-point negate	fp_neg.asm	R2::R5	none	R2::R5
fp_u8tof	convert unsigned 8-bit char to float	fp_uctof.asm	R2	none	R2::R5
fp_u16tof	convert unsigned 16-bit int to float	fp_utof.asm	R2:R3	none	R2::R5
fp_u32tof	convert unsigned 32-bit long to float	fp_ultof.asm	R2::R5	none	R2::R5

Note: For fp_cmp, the outputs represent the TMS370/C8 status bits.

5.9 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, c_int00. The runtime-support source library, rts.src, contains the source to this routine in a module named boot.c.

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

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

1) Defines a section called .sstack and sets up the software stack pointer (STK) to point to the end of the section.

2) Defines a section called .hstack for the TMS370 hardware stack area and sets up the hardware stack pointer (SP). For the TMS370C8, the hardware stack page bits in system control register 1 are set according to the location of the .hstack section.

3) Autoinitializes global variables by copying the data from the initialization tables in the .cinit section to the storage allocated for the variables in either the .bss or .reg sections. In the RAM initialization model, a loader performs this step before the program runs (it is not performed by the boot routine).

4) Calls the function main to begin running the C program.

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

See subsection 5.1.3, page 5-6, for information on how to change the default stack size.

5.9.1 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 used to initialize all the system variables. (Do not place any other data in the .cinit section; this corrupts the tables.)

Note: Initializing Variables

In standard C, global and static variables that are not explicitly initialized are set to 0 before program execution. The C compiler does not perform any pre-initialization of uninitialized variables. Any variable that must have an initial value of 0 must be explicitly initialized. The easiest method is to have a loader clear the .bss section before the program starts running or to set a fill value of 0 in the linker control map for the .bss and .reg sections.

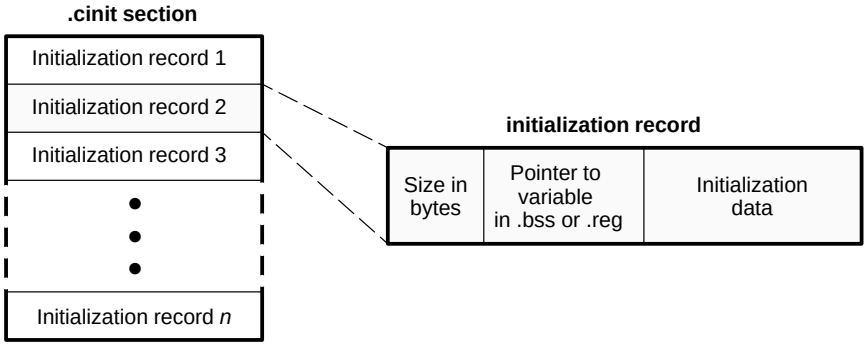
These methods (also discussed in Chapter 2) cannot be used with code that will be burned into ROM.

The two methods for copying the initialization data into memory are called the RAM and ROM models of initialization, described on pages 5-39 and 5-40, respectively.

5.9.2 Initialization Tables

The tables in the .cinit section consist of variable-size initialization records. Each variable that must be autoinitialized has an initialization record in the .cinit section. Figure 5–6 shows the format of the .cinit section and the initialization records.

Figure 5–6. Format of Initialization Records in the .cinit Section



An initialization record contains the following information:

- The first field contains the size in bytes of the initialization data.
- The second field contains the starting address of the area within the .bss or .reg section where the initialization data must be copied. (This field points to a variable's space in .bss or .reg.)
- The third field contains the data that is copied into the variable to initialize it.

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

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

The initialization information for these variables is:

```
   .sect ".cinit"        ; Initialization section
; * Record for variable i
   .word    1            ; Length of data (1 byte)
   .word    _i           ; Address in .bss
   .byte    23           ; Data

; * Record for variable a
   .word    5            ; Length of data (5 bytes)
   .word    _a           ; Address in .bss
   .byte    1,2,3,4,5    ; Data
```

The .cinit section must contain only initialization tables in this format. If you interface assembly language modules to your C program, 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.

5.9.3 Initializing Variables in the RAM Model

The RAM model of autoinitialization allows variables to be initialized at load time instead of at runtime. This enhances system performance by reducing boot time and by reserving the memory used by the initialization tables. To use this model, specify the –cr linker option.

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

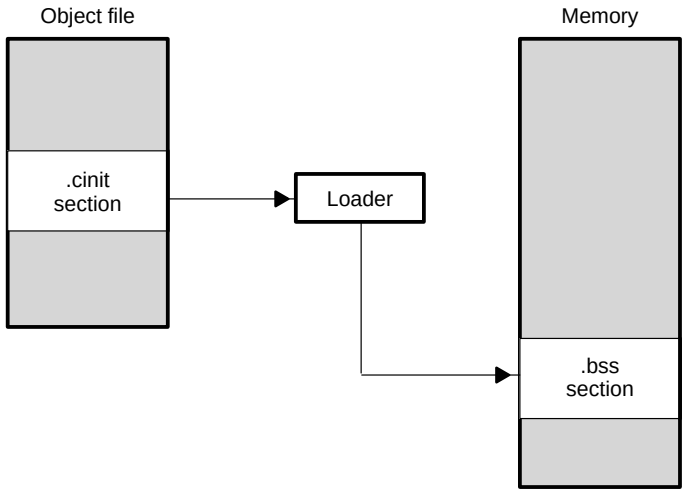
Note that the loader must be able to perform the following tasks to take advantage of the RAM model:

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

The loader then uses the initialization tables directly from the object file to initialize variables in .bss.

Figure 5–7 illustrates the RAM model of autoinitialization.

Figure 5–7. RAM Model of Autoinitialization



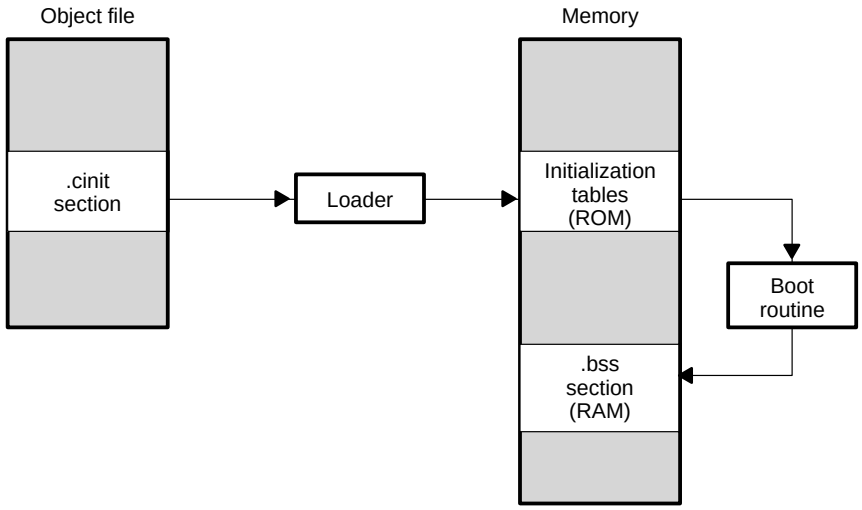
5.9.4 Initializing Variables in the ROM Model

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

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

Figure 5–8 illustrates the ROM model of autoinitialization. This method should be used in any system where your application will run from reset (that is, when your code is burned into ROM).

Figure 5–8. ROM Model of Autoinitialization



Chapter 6

Runtime-Support Functions

The runtime-support funtions included with the C compiler are standard ANSI functions that perform such tasks as memory allocation, string conversion, and string searches, which are not part of the C language. The runtime-support source library included with the C compiler, 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 use the library-build utility to build a runtime-support object library with the appropriate options, according to the device (see Chapter 7, *Library-Build Utility*); then include that library as linker input when you link your C program.

Topic **Page**

6.1 Header Files

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

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

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

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

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

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

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

Subsections 6.1.1 through 6.1.7 describe the header files that are included with the C compiler. Section 6.2, page 6-15, lists the functions that these headers declare.

6.1.1 Diagnostic Messages (assert.h)

The *assert.h* header defines the assert macro, which inserts diagnostic failure messages into programs at runtime. The assert macro tests a runtime expression. If the expression is true (nonzero), the program continues running. If the expression is false, the macro outputs a message that contains the expression, the source file name, and the line number of the statement that contains the expression; then, the program terminates (via the abort function).

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

The assert macro is defined as follows:

```
#ifndef _ASSERT
#define _ASSERT

void _nassert(int), _assert(int, char *);
#define _STR(x)  __STR(x)
#define __STR(x) #x

#if defined(NDEBUG)
#define assert(ignore) ((void)0)

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

#else
#define assert(expression)   _assert((expression),
                  \
               "Assertion failed, (" _STR(expression) "),
file " __FILE__    \
         ", line " _STR(__LINE__) "\n")
#endif
#endif
```

6.1.2 Character Typing and Conversion (ctype.h)

The *ctype.h* header declares functions that test (type) and convert characters. The character typing and converison functions are listed in Table 6–5 on page 6-16.

The character-typing functions test a character to determine whether it is a letter, a printing character, a hexadecimal digit, etc. These functions return a value of *true* (a nonzero value) or *false* (0). Character-typing functions have names in the form **is***xxx* (for example, *isdigit*).

The character conversion functions convert characters to lowercase, uppercase, or ASCII and return the converted character. Character-conversion functions have names in the form **to***xxx* (for example, *toupper*).

The ctype.h header also contains macro definitions that perform these same operations. The macros run faster than the corresponding functions. If an argument passed to one of these macros has side effects, the function version should be used instead. The typing macros expand to a lookup operation in an array of flags (this array is defined in ctype.c). The macros have the same name as the corresponding functions, but each macro is prefixed with an underscore. For example,

```
_isdigit (*p++)
```

has the effect of incrementing p. The macro can increment p more than once, but the function _isdigit assures the correct result.

6.1.3 Error Reporting (errno.h)

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.

6.1.4 Limits (float.h and limits.h)

The float.h and limits.h headers define macros that expand to useful limits and parameters of the TMS370 and 370C8's 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	8	Number of bits in type char
SCHAR_MIN	–128	Minimum value for a signed char
SCHAR_MAX	127	Maximum value for a signed char
UCHAR_MAX	255	Maximum value for an unsigned char
CHAR_MIN	–128	Minimum value for a char
CHAR_MAX	127	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

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

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	0	Rounding mode for floating-point addition
FLT_DIG DBL_DIG LDBL_DIG	6 6 6	Number of decimal digits of precision for a float, double, or long double
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	24 24 24	Number of base-FLT_RADIX digits in the mantissa of a float, double, or long double
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	–125 –125 –125	Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized float, double, or long double
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	128 128 128	Maximum positive integer such that FLT_RADIX raised to that power minus 1 is a representable finite float, double, or long double
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	1.19209290e–07 1.19209290e–07 1.19209290e–07	Minimum positive float, double, or long double number x such that $1.0 + x \neq 1.0$
FLT_MIN DBL_MIN LDBL_MIN	1.17549435e–38 1.17549435e–38 1.17549435e–38	Minimum positive float, double, or long double
FLT_MAX DBL_MAX LDBL_MAX	3.40282347e+38 3.40282347e+38 3.40282347e+38	Maximum float, double, or long double
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	–37 –37 –37	Minimum negative integers such that 10 raised to that power is in the range of normalized floats, doubles, or long doubles
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	38 38 38	Maximum positive integers such that 10 raised to that power is in the range of representable finite floats, doubles, or long doubles

Legend: FLT_ applies to type float.
DBL_ applies to type double.
LDBL_ applies to type long double.

Note: The precision of some of the values in this table has been reduced for readability. Refer to the float.h header file supplied with the compiler for the full precision carried by the processor.

6.1.5 Floating-Point Math (math.h)

The *math.h* header defines several trigonometric, exponential, and hyperbolic math functions. The math functions are listed in Table 6–5 on page 6-17. These math functions expect double-precision floating-point type arguments and return double-precision floating-point type 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.

6.1.6 Port Definitions (ports.h)

The ports.h header file defines all the port names that are associated with the ports referenced in the *TMS370 Family User's Guide* and the *TMS370C8 Central Processing Unit, System, and Instruction Set Reference Guide.* It defines these ports as extern volatile unsigned chars. Also included are macro definitions for each port. These macro names correspond to the descriptive name as used in the *TMS370 Family Data Manual* and the *TMS370C8 Central Processing Unit, System, and Instruction Set Reference Guide.*

The ports.h header includes macros DINT and EIN, which disable and enable interrupts, and macros PORT_TO_LONG and LONG_TO_PORT, which transfer data between an int variable and a '370/C8 peripheral register that is referenced by two ports. These macros receive the following arguments:

- Port MSB
- Port LSB
- Int variable (or int-sized constant value)

For example, use the LONG_TO_PORT macro to set the baud rate of the SCI module as follows:

```
LONG_TO_PORT (BAUDMSB, BAUDLSB, 129l)   ;/*129 lowercase "L"*/
```

Table 6–3 lists the TMS370 macro names defined in ports.h, their associated port name, and a brief description. Table 6–4 lists the TMS370C8 macro names, their associated port name, and a brief description.

Table 6–3. TMS370 Family Macro Names and Definitions in ports.h

Macro Name	Port Name	Description
SCCR0	_P010	System configuration control register 0
SCCR1	_P011	System configuration control register 1
SCCR2	_P012	System configuration control register 2
INT1	_P017	Interrupt 1 control register
INT2	_P018	Interrupt 2 control register
INT3	_P019	Interrupt 3 control register
DEECTL	_P01A	Data EEPROM control register
PEECTL	_P01C	Program (E)EPROM control register
APORT2	_P021	Data port A control register
ADATA	_P022	Data port A data register
ADIR	_P023	Data port A direction register
BPORT2	_P025	Data port B control register
BDATA	_P026	Data port B data register
BDIR	_P027	Data port B direction register
CPORT2	_P029	Data port C control register
CDATA	_P02A	Data port C data register
CDIR	_P02B	Data port C direction register
DPORT1	_P02C	Data port D control register 1
DPORT2	_P02D	Data port D control register 2
DDATA	_P02E	Data port D data register
DDIR	_P02F	Data port D direction register
SPICCR	_P030	SPI configuration control register
SPICTL	_P031	SPI operation control register
SPIBUF	_P037	SPI receiver buffer
SPIDAT	_P039	SPI transmitter buffer
SPIPC1	_P03D	SPI port control register 1 (SPICLK)
SPIPC2	_P03E	SPI port control register 2 (SPISIMO, SPISOMI)
SPIPRI	_P03F	SPI interrupt priority control register

Table 6–3. TMS370 Family Macro Names and Definitions in ports.h (Continued)

Macro Name	Port Name	Description
T1CNTMSB	_P040	Timer 1 counter MSB
T1CNTLSB	_P041	Timer 1 counter LSB
T1CMSB	_P042	Timer 1 compare register MSB
T1CLSB	_P043	Timer 1 compare register LSB
T1CCMSB	_P044	Timer 1 capture/compare register MSB
T1CCLSB	_P045	Timer 1 capture/compare register LSB
WDCNTMSB	_P046	Watchdog counter MSB
WDCNTLSB	_P047	Watchdog counter LSB
WDRST	_P048	Watchdog key
T1CTL1	_P049	Timer 1 counter control register 1
T1CTL2	_P04A	Timer 1 counter control register 2
T1CTL3	_P04B	Timer 1 counter control register 3
T1CTL4	_P04C	Timer 1 counter control register 4
T1PC1	_P04D	Timer 1 port control register 1
T1PC2	_P04E	Timer 1 port control register 2
T1PRI	_P04F	Timer 1 interrupt priority control reg
SCICCR	_P050	SCI communications control register
SCICTL	_P051	SCI control register
BAUDMSB	_P052	Baud rate select MSB
BAUDLSB	_P053	Baud rate select LSB
TXCTL	_P054	Transmitter int. control and status register
RXCTL	_P055	Receiver int. control and status register
RXBUF	_P057	Receiver data buffer
TXBUF	_P059	Transmit data buffer
SCIPC1	_P05D	Port control register 1
SCIPC2	_P05E	Port control register 2
SCIPRI	_P05F	Interrupt priority control register

Table 6–3. TMS370 Family Macro Names and Definitions in ports.h (Continued)

Macro Name	Port Name	Description
T2CNTMSB	_P060	Timer 2 counter MSB
T2CNTLSB	_P061	Timer 2 counter LSB
T2CMSB	_P062	Timer 2 compare register MSB
T2CLSB	_P063	Timer 2 compare register LSB
T2CCMSB	_P064	Timer 2 capture/compare register MSB
T2CCLSB	_P065	Timer 2 capture/compare register LSB
T2ICMSB	_P066	Timer 2 input capture register MSB
T2ICLSB	_P067	Timer 2 input capture register LSB
T2CTL1	_P06A	Timer 2 control register 1
T2CTL2	_P06B	Timer 2 control register 2
T2CTL3	_P06C	Timer 2 control register 3
T2PC1	_P06D	Timer 2 port control register 1
T2PC2	_P06E	Timer 2 port control register 2
T2PRI	_P06F	Timer 2 interrupt priority control reg
ADCTL	_P070	A/D control register
ADSTAT	_P071	A/D status & interrupt register
ADDATA	_P072	A/D analog data register
ADIN	_P07D	A/D digital input register
ADENA	_P07E	A/D digital input enable register
ADPRI	_P07F	A/D interrupt priority register

Table 6–4. TMS370C8 Macro Names and Definitions in ports.h

Macro Name	Port Name	Description
SCR0	P018	System control register 0
SCR1	P019	System control register 2
SRSR	P01A	System reset status register
SSR	P01B	System status register
PSA1	P01E	Parallel signature analysis 1
PSA2	P01F	Parallel signature analysis 2
DCR1	P020	Digital control register 1
DCR2	P021	Digital control register 2
DCR3	P022	Digital control register 3
DCR4	P023	Digital control register 4
DSR1	P024	Digital status register 1
DSR2	P025	Digital status register 2
DSR3	P026	Digital status register 3
DSR4	P027	Digital status register 4
ADIR	P028	Port A direction register
ADATA	P029	Port A data register
BDIR	P02A	Port B direction register
BDATA	P02B	Port B data register
CDIR	P02C	Port C direction register
CDATA	P02D	Port C data register
DDIR	P02E	Port D direction register
DDATA	P02F	Port D data register
INT1	P080	Type A interrupt
INT1 FLG	P081	Type A interrupt flag
INT2	P082	Type B interrupt
INT2 FLG	P083	Type B interrupt flag
INT3	P084	Type C interrupt
INT3 FLG	P085	Type C interrupt flag
PM2 ENABLE	P08C	Power module enable register 2
PM2 FLAGS	P08D	Power module flag register 2
PM1 ENABLE	P08E	Power module enable register 1
PM1 FLAGS	P08F	Power module flag register 1

6.1.7 Nonlocal Jumps (setjmp.h)

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

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

6.1.8 Variable Arguments (stdarg.h)

Some functions can have a variable number of arguments whose types can differ; such a function is called a *variable-argument function*. The *stdarg.h* header declares three macros and a type that help you to use variable-argument functions: *va_start*, *va_arg*, and *va_end*. These macros are used when the number and type of arguments may vary each time a function is called.

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

A variable-argument function can use the macros declared by stdarg.h to step through its argument list at run time when the function that is using the macro knows the number and types of arguments actually passed to it.

6.1.9 Standard Definitions (stddef.h)

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

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

The macros include:

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

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

6.1.10 General Utilities (stdlib.h)

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

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

The macro, *RAND_MAX*, is the maximum random number the rand function will return.

The header also declares many of the common library functions:

- **Memory management** functions that allow you to allocate and deallocate packets of memory. By default, these functions can use 512 bytes of memory. You can change this amount at link time by invoking the linker with the –heap option and specifying the desired heap size as a constant directly after the option.
- **String conversion** functions that convert strings to numeric representations
- **Searching** and **sorting** functions that allow you to search and sort arrays
- **Sequence-generation** functions that allow you to generate a pseudo-random sequence and allow you to choose a starting point for a sequence
- **Program-exit** functions that allow your program to terminate normally or abnormally
- **Integer arithmetic** that is not provided as a standard part of the C language

The general utilities functions and macros are listed in Table 6–5 on page 6-18.

6.1.11 String Functions (string.h)

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

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

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

When you use functions that move or copy strings, be sure that the destination is large enough to contain the result. The string functions are listed in Table 6–5 on page 6-19.

6.1.12 Time Functions (time.h)

The *time.h* header declares one macro, several types, and functions that manipulate dates and times. The time functions and macros are listed in Table 6–5 on page 6-20. Times are represented in two ways:

- As an arithmetic value of type *time_t*. When expressed in this way, a time is represented as a number of seconds since 12:00 AM January 1, 1900. The time_t type is a synonym for the type unsigned long.
- As a structure of type *struct_tm*. This structure contains members for expressing time as a combination of years, months, days, hours, minutes, and seconds. A time represented like this is called broken-down time. The structure has the following members.

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

A time, whether represented as a time_t or a struct_tm, can be expressed from different points of reference:

- Calendar time represents the current Gregorian date and time.
- Local time is the calendar time expressed for a specific time zone.

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

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

Note: Writing Your Own Clock Function

The clock function is host-system specific, so you must write your own clock function. You must also define the CLK_TCK macro according to the granularity 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.

6.2 Runtime-Support Functions and Macros

Table 6–5 lists the header files in the runtime-support libraries with their functions and macros. The table includes basic syntax and short descriptions for each function or macro. For information about a specific type of function or macro, use the following page references:

Table 6–5. Summary of Runtime-Support Functions and Macros

Error Message Macro (assert.h)†	Description
void **assert**(int expression);	Inserts diagnostic messages into programs
Character-Typing Conversion Functions (ctype.h) ‡	**Description**
int **isalnum(**char c);	Tests c to see if it's an alphanumeric ASCII character
int **isalpha**(char c);	Tests c to see if it's an alphabetic ASCII character
int **isascii**(char c);	Tests c to see if it's an ASCII character
int **iscntrl**(char c);	Tests c to see if it's a control character
int **isdigit**(char c);	Tests c to see if it's a numeric character
int **isgraph**(char c);	Tests c to see if it's any printing character except a space
int **islower**(char c);	Tests c to see if it's a lowercase alphabetic ASCII character
int **isprint**(char c);	Tests c to see if it's a printable ASCII character (including spaces)
int **ispunct**(char c);	Tests c to see if it's an ASCII punctuation character
int **isspace**(char c);	Tests c to see if it's an ASCII spacebar, tab (horizontal or vertical), carriage return, formfeed, or newline character
int **isupper**(char c);	Tests c to see if it's an uppercase alphabetic ASCII character
int **isxdigit**(char c);	Tests c to see if it's a hexadecimal digit
int **toascii**(char c);	Masks c into a legal ASCII value
int **tolower**(int char c);	Converts c to lowercase if it's uppercase
int **toupper**(int char c);	Converts c to uppercase if it's lowercase
Floating-Point Math Functions (math.h)	**Description**
double **acos**(double x);	Returns the arc cosine of x
double **asin**(double x);	Returns the arc sine of x
double **atan**(double x);	Returns the arc tangent of x
double **atan2**(double y, double x);	Returns the inverse tangent of y/x
double **ceil**(double x); ‡	Returns the smallest integer greater than or equal to x

† Macro

‡ Expands inline if –x is used

Table 6–5. Summary of Runtime-Support Functions and Macros (Continued)

Floating-Point Math Functions (continued)	Description
double **cos**(double x);	Returns the cosine of x
double **cosh**(double x);	Returns the hyperbolic cosine of x
double **exp**(double x); §	Returns the exponential function of x
double **fabs**(double x);	Returns the absolute value of x
double **floor**(double x); ‡	Returns the largest integer less than or equal to x
double **fmod**(double x, double y); ‡	Returns the floating-point remainder of x/y
double **frexp**(double value, int *exp);	Breaks value into a normalized fraction and an integer power of 2
double **ldexp**(double x, int exp);	Multiplies x by an integer power of 2
double **log**(double x);	Returns the natural logarithm of x
double **log10**(double x);	Returns the base-10 (common) logarithm of x
double **modf**(double value, int *iptr);	Breaks value into into a signed integer and a signed fraction
double **pow**(double x, double y);	Returns x raised to the power y
double **sin**(double x);	Returns the sine of x
double **sinh**(double x);	Returns the hyperbolic sine of x
double **sqrt**(double x);	Returns the nonnegative square root of x
double **tan**(double x);	Returns the tangent of x
double **tanh**(double x);	Returns the hyperbolic tangent of x
Variable-Argument Functions and Macros (stdarg.h)	**Description**
type **va_arg**(va_list ap); †	Accesses the next argument of type *type* in a variable-argument list
void **va_end**(va_list ap); †	Resets the calling mechanism after using va_arg
void **va_start**(va_list ap); †	Initializes ap to point to the first operand in the variable-argument list
Nonlocal Jumps Macro and Function (setjmp.h)	**Description**
int **setjmp**(jmp_buf env); †	Saves calling environment for later use by longjmp function
void **longjmp**(jmp_buf env, int returnval);	Uses jmp_buf argument to restore a previously saved program environment

† Macro
‡ Expands inline if –x is used
§ Expands inline unless –x0 is used

Table 6–5. Summary of Runtime-Support Functions and Macros (Continued)

General Utilities (stdlib.h)	Description
int **abs**(int j); §	Returns the absolute value of j
void **abort**(void)	Terminates a program abnormally
void **atexit**(void (*fun)(void));	Registers the function pointed to by fun, to be called without arguments at normal program termination
double **atof**(const char *nptr);	Converts a string to a floating-point value
int **atoi**(const char *nptr);	Converts a string to an integer value
long **atol**(const char *nptr);	Converts a string to a long integer value
void ***bsearch**(const void *key, const void *base, size_t nmemb, size_t_size, int (*_compar) (const void *, const void *));	Searches through an array of nmemb objects for the object that key points to
void ***calloc** (size_t nmemb, size_t size);	Allocates and clears memory for n objects, each of size bytes
div_t **div**(int numer, int denom);	Divides numer by denom
void **exit**(int status);	Terminates a program normally
void **free**(void *ptr);	Deallocates memory space allocated by malloc, calloc, or realloc
long **labs**(long j);	Returns the absolute value of j
ldiv_t **ldiv** (long numer, long denom);	Divides numer by denom
int **ltoa**(long n, char *buffer);	Converts n to the equivalent string
void ***malloc**(size_t size);	Allocates memory for an object of size bytes
void **minit**(void);	Resets all the memory previously allocated by malloc, calloc, or realloc
reentrant void **qsort**(void *_base, size_t nmemb, size_t _size, int (*_compar) (void *, void *));	Sorts an array of n members; base points to the first member of the unsorted array, and size specifies the size of each member
int **rand**(void);	Returns a sequence of pseudorandom integers in the range 0 to RAND_MAX
void ***realloc**(void *ptr, size_t size);	Changes the size of an allocated memory space
void **srand**(unsigned int seed);	Resets the random number generator
double **strtod**(const char *nptr, char **endptr);	Converts a string to a floating-point value
long **strtol**(const char *nptr, char **endptr, int base);	Converts a string to a long integer
unsigned long **strtoul**(const char *nptr, char **endptr**,** int base**);**	Converts a string to an unsigned long integer

Table 6–5. Summary of Runtime-Support Functions and Macros (Continued)

String Functions (string.h)	Description
void ***memchr**(void *s, int c, size_t n); ‡	Finds the first occurrence of c in the first n characters of s
int **memcmp**(void *s1, void *s2, size_t n); ‡	Compares the first n characters of s1 to s2
void ***memcpy**(void *s1, void *s2, size_t n);	Copies n characters from s2 to s1
void ***memmove**(void *s1, void *s2, size_t n);	Moves n characters from s2 to s1
void ***memset**(void *s, int c, size_t n); ‡	Copies the value of c into the first n characters of s
char ***strcat**(char *s1, char *s2); ‡	Appends s2 to the end of s1
char ***strchr**(char *s, int c); ‡	Finds the first occurrence of character c in s
int **strcmp**(char *s1, char *s2); ‡	Compares strings and returns one of the following values: <0 if s1 is less than s2; =0 if s1 is equal to s2; >0 if s1 is greater than s2
int ***strcoll**(char *s1, char *s2);	Compares strings and returns one of the following values, depending on the locale: <0 if s1 is less than s2; =0 if s1 is equal to s2; >0 if s1 is greater than s2
char ***strcpy**(char *s1, char *s2); ‡	Copies string s2 into s1
size_t **strcspn**(char *s1, char *s2);	Returns the length of the initial segment of s1 that is made up entirely of characters that are not in s2
char ***strerror**(int errnum);	Maps the error number in errnum to an error message string
size_t **strlen**(char *s); ‡	Returns the length of a string
char ***strncat**(char *s1, char *s2, size_t n);	Appends up to n characters from s1 to s2
int **strncmp**(char *s1, char *s2, size_t n); ‡	Compares up to n characters in two strings
char ***strncpy**(char *s1, char *s2, size_t n); ‡	Copies up to n characters of s2 to s1
char ***strpbrk**(char *s1, char *s2);	Locates the first occurrence in s1 of *any* character from s2
char ***strrchr**(char *s1, char c); ‡	Finds the last occurrence of character c in s
size_t **strspn**(char *s1, char *s2);	Returns the length of the initial segment of s1, which is entirely made up of characters from s2
char ***strstr**(char *s1, char *s2);	Finds the first occurrence of s2 in s1
char ***strtok**(char *s1, char *s2);	Breaks s1 into a series of tokens, each delimited by a character from s2

‡ Expands inline if –x is used

Table 6–5. Summary of Runtime-Support Functions and Macros (Continued)

Time Functions (time.h)	Description
char ***asctime**(const struct tm *timeptr);	Converts a time to a string
clock_t **clock**(void);	Determines the processor time used
char ***ctime**(const time_t *timeptr);	Converts calendar time to local time
double **difftime**(time_t time1, time_t time0);	Returns the difference between two calendar times
struct tm ***gmtime**(const time_t *timer);	Converts calendar time to Greenwich Mean Time
struct tm ***localtime**(const time_t *timer);	Converts calendar time to local time
time_t **mktime**(struct tm *timeptr);	Converts local time to calendar time
size_t **strftime**(char *s, size_t maxsize, const char *format, const struct tm *timeptr);	Formats a time into a character string
time_t **time**(time_t *timer);	Returns the current calendar time

6.3 Functions Reference

The remainder of this chapter is a reference for all runtime-support functions and macros.

6.4 Alphabetical Summary of Runtime-Support Functions

abort — *Abort*

Syntax

#included<stdlib.h>

void abort(void**);**

Defined in

exit.c in rts.src

Description

The abort function causes the program to terminate:

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

See the exit function on page 6-31.

abs/labs — *Absolute Value*

Syntax

#include <stdlib.h>

int abs(int j**);**
long int labs(long int k**);**

Defined in

abs.c and labs.c in rts.src

Description

The C compiler supports two functions that return the absolute value of an integer:

- The abs function returns the absolute value of an integer j.
- The labs function returns the absolute value of a long integer k.

acos — *Arc Cosine*

Syntax

#include <math.h>

double acos(double x**);**

Defined in

asin.c in rts.src

Description

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

Example

```
double realval, radians;

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

asctime — *Internal Time to String*

Syntax

#include <time.h>

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

Defined in

asctime.c in rts.src

Description

The asctime function converts a broken-down time into a string with the following form:

```
Mon Jan 11 11:18:36 1988 \n\0
```

The function returns a pointer to the converted string.

For more information about the functions and types that the time.h header declares, refer to subsection 6.1.12 on page 6-13.

asin — *Arc Sine*

Syntax

#include <math.h>

double asin(double x**);**

Defined in

asin.c in rts.src

Description

The asin function returns the arc sine of a floating-point argument x. x must be in the range [–1,1]. The return value is an angle in the range [–π/2,π/2] radians.

Example

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

assert — *Insert Diagnostic Information Macro*

Syntax

#include <assert.h>

void assert(int expression**);**

Defined in

assert.h as macro

Description

The assert macro tests an expression; depending upon the value of the expression, assert either issues a message and aborts execution or continues execution. This macro is useful for debugging.

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

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

```
#define assert(ignore)
```

Example

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

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

atan — *Polar Arc Tangent*

Syntax

#include <math.h>

double atan(double x**);**

Defined in

atan.c in rts.src

Description

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

Example

```
double realval, radians;

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

atan2 — *Cartesian Arc Tangent*

Syntax

#include <math.h>

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

Defined in

atan.c in rts.src

Description

The atan2 function returns the 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>

void atexit(void (*fun)(void)**);**

Defined in

exit.c in rts.src

Description

The atexit function registers the function that is pointed to by *fun*, to be called without arguments at normal program termination. Up to 32 functions can be registered.

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

atof/atoi/atol *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

Three functions convert strings to numeric representations:

- The atof function converts a string into a floating-point value. Argument nptr points to the string; the string must have the following format:

  [space] [sign] digits [.digits] [e|E [sign] integer]

- The atoi function converts a string into an integer. Argument nptr points to the string; the string must have the following format:

  [space] [sign] digits

- The atol function converts a string into a long integer. Argument nptr points to the string; the string must have the following format:

  [space] [sign] digits

The *space* is indicated by 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 then *digits* that represent the integer portion of the number. The fractional part of the number follows, then the exponent, including an optional *sign*.

The first character that cannot be part of the number terminates the string.

Since int and long are functionally equivalent in ’370/C8 C, the atoi and atol functions are also functionally equivalent.

The functions do not handle any overflow resulting from the conversion.

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

The bsearch function searches through an array of nmemb objects for a member that matches the object that key points to. Argument base points to the first member in the array; size specifies the size (in bytes) of each member.

The contents of the array must be in ascending order. If a match is found, the function returns a pointer to the matching member of the array; if no match is found, the function returns a null pointer (0).

Argument compar points to a function that compares key to the array elements. The comparison function should be declared as:

```
int cmp(const void *ptr1, const void *ptr2)
```

The cmp function compares the objects that prt1 and ptr2 point to and returns one of the following values:

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

calloc — *Allocate and Clear Memory*

Syntax

#include <stdlib.h>

void *calloc(size_t nmemb, size_t size**);**

Defined in

memory.c in rts.src

Description

The calloc function allocates size bytes (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, refer to subsection 5.1.4, page 5-10.

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

The ceil function returns a floating-point number that represents the smallest integer greater than or equal to x.

Example

```
extern double ceil();
double answer;
answer = ceil(3.1415);   /* answer = 4.0 */
answer = ceil(–3.5);     /* answer = –3.0 */
```

clock — *Processor Time*

Syntax

#include <time.h>

clock_t clock(void**);**

Defined in

clock.c in rts.src

Description

The clock function determines the amount of processor time used. It returns an approximation of the processor time used by a program since the program began running. The time in seconds is the return value divided by the value of the macro CLK_TCK.

If the processor time is not available or cannot be represented, the clock function returns the value of [(clock_t) –1].

> **Note: Writing Your Own Clock Function**
>
> The clock function is host-system specific, so you must write your own clock function. You must also define the CLK_TCK macro according to the granularity 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.

For more information about the functions and types that the time.h header declares, refer to subsection 6.1.12 on page 6-13.

cos — *Cosine*

Syntax

#include <math.h>

double cos(double x**);**

Defined in

sin.c in rts.src

Description

The cos function returns the cosine of a floating-point number x. The angle x is expressed in radians. An argument with a large magnitude may produce a result with little or no significance.

Example

```
double radians, cval; /* cos returns cval */
radians = 3.1415927;
cval = cos(radians);  /* return value = –1.0 */
```

cosh — *Hyperbolic Cosine*

Syntax

#include <math.h>

double cosh(double x**);**

Defined in

cosh.c in rts.src

Description

The cosh function returns the hyperbolic cosine of a floating-point number x. A range error occurs if the magnitude of the argument is too large.

Example

```
double x, y;

x = 0.0;
y = cosh(x); /* return value = 1.0 */
```

ctime — *Calendar Time*

Syntax

#include <time.h>

char *ctime(const time_t *timer**);**

Defined in

ctime.c in rts.src

Description

The ctime function converts a calendar time (pointed to by timer) to local time in the form of a string. This is equivalent to:

```
asctime(localtime(timer))
```

The function returns the pointer returned by the asctime function.

For more information about the functions and types that the time.h header declares, see subsection 6.1.12 on page 6-13.

difftime — *Time Difference*

Syntax

#include <time.h>

double difftime(time_t time1, time_t time0**);**

Defined in

difftime.c in rts.src

Description

The difftime function calculates the difference between two calendar times, time1 minus time0. The return value expresses seconds.

For more information about the functions and types that the time.h header declares, refer to subsection 6.1.12 on page 6-13.

div/ldiv — *Division*

Syntax

#include <stdlib.h>

div_t div(int numer, denom**);**
ldiv_t ldiv(long numer, denom**);**

Defined in

div.c in rts.src

Description

Two functions support integer division by returning numer (numerator) divided by denom (denominator). You can use these functions to get both the quotient and the remainder in a single operation.

- The div function performs integer division. The input arguments are integers; the function returns the quotient and the remainder in a structure of type div_t. The structure is defined as follows:

```
typedef struct
{
   int  quot;         /*  quotient    */
   int  rem;          /* remainder    */
} div_t;
```

- The ldiv function performs long integer division. The input arguments are long integers; the function returns the quotient and the remainder in a structure of type ldiv_t. The structure is defined as follows:

```
typedef struct
{
   long int  quot;    /*  quotient    */
   long int  rem;     /* remainder    */
} ldiv_t;
```

If the division produces a remainder, the sign of the quotient is the same as the algebraic quotient, and the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient.

Because ints and longs are equivalent types in ’370/C8 C, ldiv and div are also equivalent.

exit — *Normal Termination*

Syntax

#include <stdlib.h>

void exit(int status**);**

Defined in

exit.c in rts.src

Description

The exit function terminates a program normally. All functions registered by the atexit function are called in reverse order of their registration. The exit function can accept EXIT_FAILURE as a value. (See the abort function on page 6-22).

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

The exp function returns the exponential function of real number x. The return value is the number e raised to the power x. A range error occurs if the magnitude of x is too large.

Example

```
double x, y;

x = 2.0;
y = exp(x);        /* y = 7.38, which is e**2.0 */
```

fabs — *Absolute Value*

Syntax

#include <math.h>

double fabs(double x**);**

Defined in

fabs.c in rts.src

Description

The fabs function returns the absolute value of a floating-point number, x.

Example

```
double x, y;

x = –57.5;
y = fabs(x);       /* return value = +57.5 */
```

floor — *Floor*

Syntax

#include <math.h>

double floor(double x**);**

Defined in

floor.c in rts.src

Description

The floor function returns a floating-point number that represents the largest integer less than or equal to x.

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.

Example

```
double x, y, r;

x = 11.0;
y = 5.0;
r = fmod(x, y);       /* fmod returns 1.0 */
```

free — *Deallocate Memory*

Syntax

#include <stdlib.h>

void free(void *ptr**);**

Defined in

memory.c in rts.src

Description

The free function deallocates memory space (pointed to by ptr) that was previously allocated by a malloc, calloc, or realloc call. This makes the memory space available again. If you attempt to free unallocated space, the function takes no action and returns. For more information, refer to subsection 5.1.4 on page 5-10.

Example

This example allocates ten bytes and then frees them.

```
char *x;
x = malloc(10);       /*  allocate 10 bytes   */
free(x);              /*  free 10 bytes       */
```

frexp — *Fraction and Exponent*

Syntax

#include <math.h>

double frexp(double value**,** int *exp**)**;

Defined in

frexp.c in rts.src

Description

The frexp function breaks a floating-point number into a normalized fraction and the integer power of 2. The function returns a value with a magnitude in the range [1/2,1) or 0, so that value = = $x \times 2^{exp}$. The frexp function stores the power in the int pointed to by exp. If value is 0, both parts of the result are 0.

Example

```
double fraction;
int exp;

fraction = frexp(3.0, &exp);
/* after execution, fraction is .75 and exp is 2 */
```

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.

For more information about the functions and types that the time.h header declares, refer to subsection 6.1.12 on page 6-13.

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

These functions test a single argument c to see if it is a particular type of character — alphabetic, alphanumeric, numeric, ASCII, etc. If the test is true, the function returns a nonzero value; if the test is false, the function returns 0. The character typing functions include:

isalnum	identifies alphanumeric ASCII characters (tests for any character for which isalpha or isdigit is true).
isalpha	identifies alphabetic ASCII characters (tests for any character for which islower or isupper is true).
isascii	identifies ASCII characters (any character from 0–127).
iscntrl	identifies control characters (ASCII characters 0–31 and 127).
isdigit	identifies numeric characters between 0 and 9 (inclusive).
isgraph	identifies any nonspace character.
islower	identifies lowercase alphabetic ASCII characters.
isprint	identifies printable ASCII characters, including spaces (ASCII characters 32–126).
ispunct	identifies ASCII punctuation characters.
isspace	identifies ASCII tab (horizontal or vertical), space bar, carriage return, form feed, and new line characters.
isupper	identifies uppercase ASCII alphabetic characters.
isxdigit	identifies hexadecimal digits (0–9, a–f, A–F).

The C compiler also supports a set of macros that perform these same functions. The macros have the same names as the functions but are prefixed with an underscore; for example, _isascii is the macro equivalent of the isascii function. In general, the macros execute more efficiently than the functions.

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

The ldexp function multiplies a floating-point number by the power of 2 and returns $x \times 2^{exp}$. The *exp* can be a negative or a positive value. A range error may occur if the result is too large.

Example

```
double result;

result = ldexp(1.5, 5);         /* result is 48.0 */
result = ldexp(6.0, –3);        /* result is 0.75 */
```

localtime *Local Time*

Syntax

#include <time.h>

struct tm *localtime(const time_t *timer**);**

Defined in

localtime.c in rts.src

Description

The localtime function converts a calendar time (pointed to by timer) into a broken-down time, which is expressed as local time. The function returns a pointer to the converted time.

For more information about the functions and types that the time.h header declares, refer to subsection 6.1.12 on page 6-13.

log *Natural Logarithm*

Syntax

#include <math.h>

double log(double x**);**

Defined in

log.c in rts.src

Description

The log function returns the natural logarithm of a real number x. A domain error occurs if x is negative; a range error occurs if x is 0.

Description

```
float x, y;

x = 2.718282;
y = log(x);         /* Return value = 1.0 */
```

log10 — *Common Logarithm*

Syntax

#include <math.h>

double log10(double x**);**

Defined in

log10.c in rts.src

Description

The log10 function returns the base-10 logarithm of a real number x. A domain error occurs if x is negative; a range error occurs if x is 0.

Example

```
float x, y;

x = 10.0;
y = log(x);        /* Return value = 1.0 */
```

ltoa — *Long Integer to ASCII*

Syntax

no prototype provided

int ltoa(long n, char *buffer**)**;

Defined in

ltoa.c in rts.src

Description

The ltoa function is a nonstandard (non-ANSI) function and is provided for compatibility. The standard equivalent is sprintf. The function is not prototyped in rts.src. The ltoa function converts a long integer n to an equivalent ASCII string and writes it into the buffer. If the input number n is negative, a leading minus sign is output. The ltoa function returns the number of characters placed in the buffer.

malloc — *Allocate Memory*

Syntax

#include <stdlib.h>

void *malloc(size_t size**);**

Defined in

memory.c in rts.src

Description

The malloc function allocates space for an object of size bytes and returns a pointer to the space. If malloc cannot allocate the packet (that is, if it runs out of memory), it returns a null pointer (0). This function does not modify the memory it allocates.

The memory that malloc uses is in a special memory pool or heap. 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, refer to subsection 5.1.4 on page 5-10.

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

The memchr function finds the first occurrence of c in the first n characters of the object that s points to. If the character is found, memchr returns a pointer to the located character; otherwise, it returns a null pointer (0).

The memchr function is similar to strchr, except that the object that memchr searches can contain values of 0 and c can be 0.

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

The memcmp function compares the first n characters of the object that s2 points to with the object that s1 points to. The function returns one of the following values:

<	**0**	if *s1 is less than *s2
	0	if *s1 is equal to *s2
>	**0**	if *s1 is greater than *s2

The memcmp function is similar to strncmp, except that the objects that memcmp compares can contain values of 0.

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

The memcpy function copies n characters from the object that s2 points to into the object that s1 points to. If you attempt to copy characters of overlapping objects, the function's behavior is undefined. The function returns the value of s1.

The memcpy function is similar to strncpy, except that the objects that memcpy copies can contain values of 0.

memmove — *Memory Block Copy — Overlapping*

Syntax

#include <string.h>

void *memmove(void *s1, const void *s2, size_t n**);**

Defined in

memmove.c in rts.src

Description

The memmove function moves n characters from the object that s2 points to into the object that s1 points to; the function returns the value of s1. The memmove function correctly copies characters between overlapping objects.

memset — *Duplicate Value in Memory*

Syntax

#include <string.h>

void *memset(void *s, int c, size_t n**);**

Defined in

memset.c in rts.src

Description

The memset function copies the value of c into the first n characters of the object that s points to. The function returns the value of s.

minit — *Reset Dynamic Memory Pool*

Syntax

no prototype provided

void minit(void**);**

Defined in

memory.c in rts.src

Description

The minit function resets all the space that was previously allocated by calls to the malloc, calloc, or realloc functions.

The memory that minit uses is in a special memory pool or heap. The constant _ _SYSMEM_SIZE defines the size of the heap as 2K bytes. You can change this amount at link time by invoking the linker with the –heap option and specifying the desired size of the heap (in bytes) directly after the option. For more information, refer to subsection 5.1.4, page 5-10.

Note: No Previously Allocated Objects Are Available After minit

Calling the minit function makes *all* the memory space in the heap available again. Any objects that you allocated previously will be lost; do not try to access them.

mktime — *Convert to Calendar Time*

Syntax

#include <time.h>

time_t *mktime(struct tm *timeptr**);**

Defined in

mktime.c in rts.src

Description

The mktime function converts a broken-down time, expressed as local time, into 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, refer to subsection 6.1.12 on page 6-13.

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 value**,** double *iptr**);**

Defined in

modf.c in rts.src

Description

The modf function breaks a value into a signed integer and a signed fraction. Each of the two parts has the same sign as the input argument. The function returns the fractional part of value and stores the integer as a double at the object pointed to by iptr.

Example

```
double value, ipart, fpart;

value = –3.1415;
fpart = modf(value, &ipart);

/* After execution, ipart contains –3.0, */
/* and fpart contains –0.1415.           */
```

pow — *Raise to a Power*

Syntax

#include <math.h>

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

Defined in

pow.c in rts.src

Description

The pow function returns x raised to the power y. A domain error occurs if $x = 0$ and $y \leq 0$, or if x is negative and y is not an integer. A range error may occur.

Example

```
double x, y, z;

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

qsort — *Array Sort*

Syntax

#include <stdlib.h>

reentrant void qsort(void *base, size_t n, size_t size,
int (*compar) (const void *, const void *)**)**;

Defined in

qsort.c in rts.src

Description

The qsort function sorts an array of n members. Argument base points to the first member of the unsorted array; argument size specifies the size of each member.

This function sorts the array in ascending order.

Argument compar points to a function that compares key to the array elements. The comparison function should be declared as:

```
int  cmp(const void *ptr1, const void *ptr2)
```

The cmp function compares the objects that ptr1 and ptr2 point to and returns one of the following values:

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

rand/srand — *Random Integer*

Syntax

#include <stdlib.h>

int rand(void**);**
void srand(unsigned int seed**)**;

Defined in

rand.c in rts.src

Description

Two functions work together to provide pseudorandom sequence generation:

- The rand function returns pseudorandom integers in the range 0–RAND_MAX.
- 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

The realloc function changes the size of the allocated memory pointed to by ptr to the size specified in bytes by size. The contents of the memory space (up to the lesser of the old and new sizes) is not changed.

- If ptr is 0, realloc behaves like malloc.
- If ptr points to unallocated space, the function takes no action and returns 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, refer to subsection 5.1.4 on page 5-10.

setjmp/longjmp *Nonlocal Jumps*

Syntax

#include <setjmp.h>

int setjmp(jmp_buf env**)**
void longjmp(jmp_buf env, int returnval**)**

Defined in

setjmp.asm and longjmp.asm in rts.src

Description

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

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

  If the return is from a direct invocation, the setjmp macro returns the value zero. If the return is from a call to the longjmp function, the setjmp macro returns a nonzero value.
- The **longjmp** function restores the environment that was saved in the jmp_buf argument by the most recent invocation of the setjmp macro. If the setjmp macro was not invoked, or if it terminated execution irregularly, the behavior of longjmp is undefined.

  After longjmp is completed, the program execution continues as if the corresponding invocation of setjmp had just returned the value specified by **val**. The longjmp function will not cause setjmp to return a value of zero even if val is zero. If returnval is zero, the setjmp macro returns the value 1.

Example

These functions are typically used to effect an immediate return from a deeply nested function call:

```
#include <setjmp.h>
jmp_buf env;
main()
{
   int errcode;

   if ((errcode = setjmp(env)) == 0)
      nest1();
   else
      switch (errcode)
   . . .
}
   . . .
nest42()
{
   if (input() == ERRCODE42)
   /* return to setjmp call in main */
      longjmp (env, ERRCODE42);
   . . .
}
```

sin — *Sine*

Syntax

#include <math.h>

double sin(double x**);**

Defined in

sin.c in rts.src

Description

The sin function returns the sine of a floating-point number x. The angle x is expressed in radians. An argument with a large magnitude may produce a result with little or no significance.

Example

```
double radian, sval;     /* sval is returned by sin */

radian = 3.1415927;
sval = sin(radian);      /* –1 is returned by sin  */
```

sinh — *Hyperbolic Sine*

Syntax

#include <math.h>

double sinh(double x**);**

Defined in

sinh.c in rts.src

Description

The sinh function returns the hyperbolic sine of a floating-point number x. A range error occurs if the magnitude of the argument is too large.

Example

```
double x, y;

x = 0.0;
y = sinh(x);      /* return value = 0.0 */
```

sqrt — *Square Root*

Syntax

#include <math.h>

double sqrt(double x**);**

Defined in

sqrt.c

Description

The sqrt function returns the non-negative square root of a real number x. A domain error occurs if the argument is negative.

Example

```
double x, y;

x = 100.0;
y = sqrt(x);      /* return value = 10.0 */
```

strcat — *Concatenate Strings*

Syntax

#include <string.h>

char *strcat(char *s1, char *s2**);**

Defined in

strcat.c in rts.src

Description

The strcat function appends a copy of s2 (including 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"              */
/* b --> " jumps over \0"                                 */
/* c --> "the lazy dog.\0" */
strcat (a,c);
/*a --> "The quick black fox jumps over the lazy dog.\0"*/
/* b --> " jumps over \0"                                 */
/* c --> "the lazy dog.\0"                                */
```

strchr — *Find First Occurrence of a Character*

Syntax

#include <string.h>

char *strchr(const char *s, char c**);**

Defined in

strchr.c in rts.src

Description

The strchr function finds the first occurrence of c in s. If strchr finds the character, it returns a pointer to the character; otherwise, it returns a null pointer (0).

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 and strcoll.c in rts.src

Description

The strcmp and strcoll functions compare s2 with s1. The functions are equivalent; both functions are supported to provide compatibility with ANSI C.

The functions return one of the following values:

<	**0**	if *s1 is less than *s2
	0	if *s1 is equal to *s2
>	**0**	if *s1 is greater than *s2

Example

```
char *stra = ”why ask why”;
char *strb = ”just do it”;
char *strc = ”why ask why”;

if (strcmp(stra, strb) > 0)
   {
      /*    statements here will be executed    */
   }
if (strcoll(stra, strc) == 0)
   {
      /* statements here will be executed also */
   }
```

strcpy — *String Copy*

Syntax

#include <string.h>

char *strcpy(char *s1, const char *s2**);**

Defined in

strcpy.c in rts.src

Description

The strcpy function copies s2 (including a terminating null character) into s1. If you attempt to copy strings that overlap, the function's behavior is undefined. The function returns a pointer to s1.

Example

In the following example, the strings pointed to by *a and *b are two separate and distinct memory locations. In the comments, the notation \0 represents the null character:

```
char *a = ”The quick black fox”;
char *b = ” jumps over ”;

/* a –-> ”The quick black fox\0”   */
/* b –-> ” jumps over \0”          */

strcpy(a,b);

/* a –-> ” jumps over \0”          */
/* b –-> ” jumps over \0”          */
```

strcspn — *Find Number of Unmatching Characters*

Syntax

#include <string.h>

size_t strcspn(const char *s1, const char *s2**);**

Defined in

strcspn.c in rts.src

Description

The strcspn function returns the length of the initial segment of s1, which is made up entirely of characters that are not in s2. If the first character in s1 is in s2, the function returns 0.

Example

```
char *stra = ”who is there?”;
char *strb = ”abcdefghijklmnopqrstuvwxyz”;
char *strc = ”abcdefg”;
size_t length;

length = strcspn(stra,strb);    /* length = 0 */
length = strcspn(stra,strc);    /* length = 9 */
```

strerror *String Error*

Syntax

#include <string.h>

char *strerror(int 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

The strftime function formats a time (pointed to by timeptr) according to a format string and returns the formatted time in the string s. Up to maxsize characters can be written to s. The format parameter is a string of characters that tells the strftime function how to format the time; the following list shows the valid characters and describes what each character expands to.

Character 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, refer to subsection 6.1.12 on page 6-13.

strlen *Find String Length*

Syntax

#include <string.h>

size_t strlen(const char *s**);**

Defined in

strlen.c in rts.src

Description

The strlen function returns the length of s. In C, a character string is terminated by the first byte with a value of 0 (a null character). The returned result does not include the terminating null character.

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

The strncat function 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.

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

The strncmp function compares up to n characters of s2 with s1. The function returns one of the following values:

<	**0**	if *s1 is less than *s2
	0	if *s1 is equal to *s2
>	**0**	if *s1 is greater than *s2

Example

```
char *stra = ”why ask why”;
char *strb = ”just do it”;
char *strc = ”why not?”;
size_t size = 4;

if (strcmp(stra, strb, size) > 0)
   {
      /* statements here will get executed */
   }
if (strcomp(stra, strc, size) == 0)
   {
      /* statements here will get executed also */
   }
```

strncpy — *String Copy*

Syntax

#include <string.h>

char *strncpy(const char *s1, const char *s2**,** size_t n**);**

Defined in

strncpy.c in rts.src

Description

The strncpy function copies up to n characters from s2 into s1. If s2 is n characters long or longer, the null character that terminates s2 is not copied. If you attempt to copy characters from overlapping strings, the function's behavior is undefined. If s2 is shorter than n characters, strncpy appends null characters to s1 so that s1 contains n characters. The function returns the value of s1.

Example

Note that strb contains a leading space to make it five characters long. Also note that the first five characters of strc are an "I", a space, the word "am", and another space, so that after the second execution of strncpy, stra begins with the phrase "I am" followed by two spaces. In the comments, the notation \0 represents the null character.

```
char *stra = ″she's the one mother warned you of″;
char *strb = ″ he's″;
char *strc = ″I am the one father warned you of″;
char *strd = ″oops″;
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

The strpbrk function locates the first occurrence in s1 of *any* character in s2. If strpbrk finds a matching character, it returns a pointer to that character; otherwise, it returns a null pointer (0).

Example

```
char *stra = ″it wasn’t me″;
char *strb = ″wave″;
char *a;

a = strpbrk (stra,strb);
```

After this example, *a points to the “w” in “wasn’t”.

strrchr — *Find Last Occurrence of a Character*

Syntax

#include <string.h>

char *strrchr(const char *s**,** int c**);**

Defined in

strrchr.c in rts.src

Description

The strrchr function finds the last occurrence of c in s. If strrchr finds the character, it returns a pointer to the character; otherwise, it returns a null pointer (0).

Example

```
char *a = ″When zz comes home, the search is on for zs″;
char *b;
char the_z = ’z’;
```

After this example, *b points to the “z” in “zs” near the end of the string.

strspn — *Find Number of Matching Characters*

Syntax

#include <string.h>

size_t *strspn(const char *s1, const char *s2**);**

Defined in

strspn.c in rts.src

Description

The strspn function returns the length of the initial segment of s1, which is entirely made up of characters in s2. If the first character of s1 is not in s2, the strspn function returns 0.

Example

```
char *stra = "who is there?";
char *strb = "abcdefghijklmnopqrstuvwxyz";
char *strc = "abcdefg";
size_t length;

length = 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

The strstr function finds the first occurrence of s2 in s1 (excluding the terminating null character). If strstr finds the matching string, it returns a pointer to the located string; if it doesn't find the string, it returns a null pointer. If s2 points to a string with length 0, strstr returns s1.

Example

```
char *stra = "so what do you want for nothing?";
char *strb = "what";
char *ptr;

ptr = strstr(stra,strb);
```

The pointer *ptr now points to the w in what in the first string.

strtod/strtol/ strtoul

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 int strtoul(const char *nptr, char **endptr, int base**);**

Defined in

strtod.c, strtol.c, and strtoul.c in rts.src

Description

Three functions convert ASCII strings to numeric values. For each function, argument 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 causes an overflow or an underflow, errno is set to the value of ERANGE.

- The strtol function converts a string to a long integer. The string must have the following format:

  [space] [sign] digits [.digits] [e|E [sign] integer]

- The strtoul function converts a string to an unsigned long integer. The string must be specified in the following format:

  [space] [sign] digits [.digits] [e|E [sign] integer]

The space is indicated by a horizontal or vertical tab, space bar, carriage return, form feed, or new line. Following the space is an optional sign, and then digits that represent the integer portion of the number. The fractional part of the number follows, then the exponent, including an optional sign.

The first unrecognized character terminates the string. The pointer that endptr points to is set to point to this character.

strtok — *Break String into Token*

Syntax

#include <string.h>

char *strtok(char *s1, const char *s2**);**

Defined in

strtok.c in rts.src

Description

Successive calls to the strtok function break s1 into a series of tokens, each delimited by a character from s2. Each call returns a pointer to the next token.

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”  */
```

tan — *Tangent*

Syntax

#include <math.h>

double tan(double x**);**

Defined in

tan.c in rts.src

Description

The tan function returns the tangent of a floating-point number x. The angle x is expressed in radians. An argument with a large magnitude may produce a result with little or no significance.

Example

```
double x, y;
x = 3.1415927/4.0;
y = tan(x);           /* return value = 1.0 */
```

tanh — *Hyperbolic Tangent*

Syntax

#include <math.h>

double tanh(double x**);**

Defined in

tanh.c in rts.src

Description

The tanh function returns the hyperbolic tangent of a floating-point number x.

Example

```
double x, y;
x = 0.0;
y = tanh(x);       /* return value = 0.0 */
```

time — *Time*

Syntax

#include <time.h>

time_t time(time_t *timer**);**

Defined in

time.c in rts.src

Description

The time function determines the current calendar time, represented 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, refer to subsection 6.1.12, page 6-13.

Note: The time Function Is Target-System Specific

The time function is target-system specific, so you must write your own time function.

toascii — *Convert to ASCII*

Syntax

#include <ctype.h>

int toascii(int c**);**

Defined in

toascii.c in rts.src

Description

The toascii function ensures that c is a valid ASCII character by masking the lower seven bits. There is also an equivalent macro call _toascii.

tolower/toupper — *Convert Case*

Syntax

#include <ctype.h>

int tolower(int c**);**
int toupper(int c**);**

Defined in

tolower.c and toupper.c in rts.src

Description

Two functions convert the case of a single alphabetic character c into uppercase or lowercase:

- The tolower function converts an uppercase argument to lowercase. If c is already in lowercase, tolower returns it unchanged.
- The toupper function converts a lowercase argument to uppercase. If c is already in uppercase, toupper returns it unchanged.

The functions have macro equivalents named _tolower and _toupper.

va_arg/va_end/ va_start

Variable-Argument Macros

Syntax

#include <stdarg.h>

typedef char ***va_list;**
va_arg(ap, *type***);**
void va_end(ap);
void va_start(ap, *parmN***);**
va_list *ap

Defined in

stdarg.h

Description

Some functions can be called with a varying number of arguments that have varying types. Such a function, called a **variable-argument function**, can use the following macros to step through its argument list at runtime. The ap parameter points to an argument in the variable-argument list.

- The va_start macro initializes ap to point to the first argument in an argument list for the variable-argument function. The parmN parameter points to the right-most parameter in the fixed, declared list.
- The va_arg macro returns the value of the next argument in a call to a variable-argument function. Each time you call va_arg, it modifies ap so that successive arguments for the variable-argument function can be returned by successive calls to va_arg (va_arg modifies ap to point to the next argument in the list). The type parameter is a type name; it is the type of the current argument in the list.
- The va_end macro resets the stack environment after va_start and va_arg are used.

Note that you must call va_start to initialize ap before calling va_arg or va_end.

Example

```
int        printf (char *fmt...)
           va_list ap;
           va_start(ap, fmt);
                      .
                      .
                      .
/* Get next arg, an integer      */
           i = va_arg(ap, int);
/* Get next arg, a string        */
           s = va_arg(ap, char *);
/* Get next arg, a long          */
           l = va_arg(ap, long);
                      .
                      .
                      .
           va_end(ap) /* Reset   */
}
```

Chapter 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 object libraries, this package includes the source archive, rts.src, that contains source code for all runtime-support functions. You can build your own runtime-support libraries using selected options by using the mk370 utility described in this chapter and the archiver described in the *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide*.

Topics covered in this chapter include:

Topic **Page**

7.1 Invoking the Library-Build Utility

The general syntax for invoking the library-build utility is:

mk370 [*options*] *src_arch1* [**–l***obj.lib1*] [*src_arch2* [**–l***obj.lib2*]] ...

mk370 is the command that invokes the utility.

options can appear anywhere on the command line or in a linker command file. (Options are discussed in Section 7.2 and below.)

src_arch is the name of a source archive file. For each source archive named, mk370 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, mk370 uses the name of the source archive and appends a *.lib* suffix. For each source archive file specified, a corresponding object library file is created. An object library cannot be built from multiple source archive files.

The mk370 utility runs the shell program on each source file in the archive to compile and/or assemble it. Then, the utility collects all the object files into the object library. All the tools must be in your PATH environment variable. The utility ignores the environment variables TMP, C_OPTION, and C_DIR.

7.2 Options Summary

Some of the options that can be used with the library-build utility correspond directly to the options that the compiler or assembler use. These options are described in detail in Section 2.2 on page 2-7. Table 7–1 provides a summary of the options that you can use with the utility.

Table 7–1. Summary of Library-Build Options and Their Effects

(a) Options that control the library-build utility

Option	Effect
– –c	extract C source files contained in the source archive from the library and leave them in the current directory after the utility completes execution
– –h	instruct mk370 to use header files contained in the source archive and leave them in the current directory after the utility completes execution. You will probably want to use this option to install the runtime-support header files from the rts.src archive that is shipped with the tools.
– –k	instruct the utility 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	instruct the utility to suppress header information (quiet)
– –u	instruct mk370 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	print progress information to the screen during execution of the utility. Normally, the utility operates silently (no screen messages).

(b) Option that controls the compiler shell

Option	Effect	Pages
–g	enable symbolic debugging	2-14

Table 7–1. Summary of Library-Build Options and Their Effects (Continued)

(c) Options that control the parser

Option	Effect	Pages
–pk	make code K&R compatible	4-21
–pw2	enable all warning messages	2-43
–pw1	enable serious warning messages (default)	2-43
–pw0	disable all warning messages	2-43
–p?	enable trigraph expansion	2-23

(d) Options that control the level of optimization

Option	Effect	Pages
–o0	compile with register optimization	2-24
–o1	compile with –o0 optimization + local optimization	2-24
–o2 (or –o)	compile with –o1 optimization + global optimization	2-25
–o3	compile with –o2 optimization + file optimization Note that mk370 automatically sets –oI0 and –op0.	2-25

(e) Options that control the definition-controlled inline function expansion

Option	Effect	Pages
–x1	enable intrinsic function inlining	2-37
–x2 (or –x)	define _INLINE + above + invoke optimizer (at –o2 if not specified differently)	2-37

(f) Options that change the C runtime model

Option	Effect	Pages
–ma	assume variables are aliased	2-15
–mc	disable generation of overlay directives	2-15
–mi	disable generation of .ic calls directives when an indirect call is encountered	2-15
–mn	enable optimizer options disabled by –g	2-15
–mu	prevent optimizing unsigned 8-bit loop induction variables	2-15

Table 7–1. Summary of Library-Build Options and Their Effects (Continued)

(g) Options that relax type checking

Option	Effect	Pages
–tf	relax prototype checking	2-17
–tp	relax pointer combination checking	2-17

(h) Option that controls the assembler

Option	Effect	Pages
–as	keep labels as symbols	2-18

(i) Options that change the default file extensions

Option	Effect	Pages
–ea	set default extension for assembly files	2-5
–eo	set default extension for object files	2-5

(j) Option that generates TMS370C8 code

Option	Effect	Pages
–c8	generate TMS370C8 code	2-14

Appendix A

Optimizations

The TMS370/C8 C compiler uses a variety of optimization techniques to improve the execution speed of your C programs and to reduce their size. Optimization occurs at various levels throughout the compiler. Most of the optimizations described here are performed by the separate optimizer pass that you enable and control with the –o compiler options. (For details about the –o options, refer to Section 2.4, page 2-24.) However, the code generator performs some optimizations, particularly the TMS370/C8-specific optimizations, which you cannot selectively enable or disable.

This appendix describes general and TMS370/C8-specific optimizations that are performed throughout the compiler.

Topic **Page**

A.1 General Optimizations

Following are the general optimizations, which improve any C code:

- Alias disambiguation
- Branch optimizations, control-flow simplification
- Data flow optimizations
  - Copy propagation
  - Common subexpression elimination
  - Redundant assignment elimination
- Expression simplification
- Inline expansion of runtime-support library functions
- Loop-induction variable optimizations, strength reduction
- Loop-invariant code motion
- Loop rotation
- Tail merging

A.1.1 Alias Disambiguation

Programs written in the C language generally use many pointer variables. Frequently, compilers are unable to determine whether or not two or more l (lowercase L) values (symbols, pointer references, or structure references) refer to the same memory location. This *aliasing* of memory locations often prevents the compiler from retaining values in registers because it cannot be sure that the register and memory continue to hold the same values over time. Alias disambiguation is a technique that determines when two pointer expressions cannot point to the same location, allowing the compiler to freely optimize such expressions.

A.1.2 Branch Optimizations / Control-Flow Simplification

The compiler analyzes the branching behavior of a program and rearranges the linear sequences of operations (basic blocks) to remove branches or redundant conditions. Unreachable code is deleted, branches to branches are bypassed, and conditional branches over unconditional branches are simplified to a single conditional branch. When the value of a condition can be determined at compile time (through copy propagation or other data flow analysis), a conditional branch can be deleted. Switch case lists are analyzed in the same way as conditional branches and are sometimes eliminated entirely. Some simple control flow constructs can be reduced to conditional instructions, totally eliminating the need for branches. In Example A–1, 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–1. Control-Flow Simplification and Copy Propagation

(a) C source

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

(b) TMS370/C8 C compiler output

```
;   opt370 –s –O3 fsm.if fsm.opt
_fsm:
;* B     assigned to _input
L2:
;***    ----------------------g2:
;*** 9  ----------------------    if ( *input++ == 0 && *input++ ) goto g2;
;>>>>              case ALPHA: state = ( *input++ ==0) ? BETA: GAMMA; break;
;>>>>              case BETA: state = ( *input++ ==0) ? GAMMA: ALPHA; break;
         INCW      #2,R2
         MOV       *–1[R2],A
         MOV       A,R4
         MOV       *–2[R2],A
         MOV       A,R3
         OR        R4,R3
         JNE       L6
         INCW      #2,R2
         MOV       *–1[R2],A
         MOV       A,R4
         MOV       *–2[R2],A
         MOV       A,R3
         OR        R4,R3
         JNE       L2
         JMP       L6
;*** 9  ----------------------    goto g4;
L5:
;***    ----------------------g3:
;*** 11 ----------------------    input += 2;
;>>>>              case GAMMA: state = ( *input++ ==0) ? GAMMA: OMEGA; break;
         INCW      #2,R2
L6:
;***    ----------------------g4:
;*** 11 ----------------------    if ( !*input ) goto g3;
         MOV       *1[R2],A
         MOV       A,R4
         MOV       *R2,A
         MOV       A,R3
         OR        R4,R3
         JEQ       L5
;***    ----------------------    return;
         RTS
```

A.1.3 Data Flow Optimizations

Collectively, the following three data flow optimizations replace expressions with less costly ones, detect and remove unnecessary assignments, and avoid operations that produce values already computed. The optimizer performs these data flow optimizations both locally (within basic blocks) and globally (across entire functions). See Example A–1 and Example A–2.

- **Copy propagation**

  Following an assignment to a variable, the compiler replaces references to the variable with its value. The value could be another variable, a constant, or a common subexpression. This may result in increased opportunities for constant folding, common subexpression elimination, or even total elimination of the variable. See Example A–1 and Example A–2.

- **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–2.

- **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–2.

A.1.4 Expression Simplification

For optimal evaluation, the compiler simplifies expressions into equivalent forms requiring fewer instructions or registers. For example, the expression (a + b) – (c + d) takes six instructions to evaluate; it can be optimized to ((a + b) – c) – d, which takes only four instructions. Operations between constants are folded into single constants. For example, a = (b + 4) – (c + 1) becomes a = b – c + 3. See Example A–2.

Example A–2. Data Flow Optimizations and Expression Simplification

(a) C source

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

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

(b) TMS370/C8 C compiler output

```
;   opt370 –s –O3 simp.if simp.opt
;***************************************************************
;* FUNCTION DEF: _simp                                         *
;***************************************************************
_simp:
;* R2    assigned to _j
;* R4    assigned to C$2
;* R10   assigned to C$1
;* R2    assigned to _j
;>>>>      char a = 3;
;>>>>      char b = (j * a) + (j * 2);
;>>>>      char c = (j << a);
;>>>>      char d = (j >> 3) + (j << b);
;*** 8 ----------------------    C$2 = (int)j;
;>>>>      call (a,b,c,d);
         MOV       R2,R5
         CMP       #128,R5
         SBB       R4,R4
         INV       R4
;*** 8 ----------------------    C$1 = (char)(C$2*5);
         MOV       R5,R11
         RL        R11
         RL        R11
         AND       #252,R11
         ADD       R5,R11
         CMP       #128,R11
         SBB       R10,R10
         INV       R10
;*** 8 ----------------------   call(3,C$1,(int)(char)(C$2<<3),
(int)(char)((C$2>>3)+(C$2<<C$1)));
         MOV       R11,R3
         CALL      shl8
         .calls    _simp, shl8
         MOV       R2,R7
         MOVW      R5,R3
         MOV       #3,R4
         CALL      ashr16
         .calls    _simp, ashr16
         ADD       R7,R3
         MOV       R3,R9
         CMP       #128,R9
         SBB       R8,R8
         INV       R8
         MOV       R5,R7
         RL        R7
         RL        R7
         RL        R7
         AND       #248,R7
         CMP       #128,R7
         SBB       R6,R6
         INV       R6
         MOVW      #3,R3
         MOVW      R11,R5
         CALL      _call
         .calls    _simp, _call
;***   ----------------------    return;
         RTS
```

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.

A.1.5 Inline Expansion of Runtime-Support Library Functions

The compiler replaces calls to small runtime-support 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–3.

Example A–3. Inline Function Expansion

(a) C source

```
# include <string.h>
# define MAX_LEN 4

proc_ans (char *ans)
{
   char buf[MAX_LEN];
   memcpy (buf, ans, 3);
}
```

(b) TMS370/C8 C compiler output

```
;***************************************************
;* FUNCTION DEF: _proc_ans
;***************************************************
_proc_ans:
;* R2    assigned to _ans
;* R4    assigned to _ans
;>>>>     char buf[MAX_LEN];
         MOVW      R3,R5
;*** 7 ----------------------    memcpy(&buf, (void*)ans,
3u);
;>>>>     memcpy (buf, ans, 3);
         MOVW      #$B_proc_ans+0,R3
         MOVW      #3,R7
         CALL      _memcpy
         .calls    _proc_ans, _memcpy
;***   ----------------------    return;
         RTS
```

The compiler finds the code for the C function strcpy( ) in the inline library and copies it in place of the call.

A.1.6 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. 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–3, page A-6, and Example A–6, page A-11.

A.1.7 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–6, page A-11.

A.1.8 Loop Rotation

The compiler evaluates loop conditionals at the bottom of loops, saving a costly extra branch out of the loop. In many cases, the initial entry conditional check and the branch are optimized out.

A.1.9 Tail Merging

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

In Example A–4, the addition to a at the end of all three cases is merged into one block. Also, the multiplication by 3 in the first two cases is merged into one block. This results in a reduction of three instructions. Note that in some cases this optimization adversely affects execution speed because extra branches may be introduced.

Example A–4. Tail Merging

(a) C source

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

(b) TMS370/C8 C compiler output

```
_main:
         MOVW      R3,$R_main+1
         CMP       #0,$R_main+0
         JG        L6
         JL        L4
         CMP       #0,$R_main+1
         JHS       L6
L4:
         INV       $R_main+0
         INV       $R_main+1
         INCW      #1,$R_main+1
         MOVW      $R_main+1,R3
L5:
         CALL      _f
         .calls    _main, _f
         MOVW      R3,B
         RLC       R3
         RLC       R2
         ADD       B,R3
         ADC       A,R2
         JMP       L9
L6:
         MOV       $R_main+0,A
         OR        $R_main+1,A
         JNE       L8
         MOVW      $R_main+1,R3
         CALL      _g
         .calls    _main, _g
         MOVW      R3,$R_main+1
         JMP       L5
L8:
         MOVW      $R_main+1,R3
         CALL      _f
         .calls    _main, _f
L9:
         ADD       R3,$R_main+1
         ADC       R2,$R_main+0
         MOVW      $R_main+1,R3
         RTS
```

A.2 TMS370/C8-Specific Optimizations

These optimizations are designed especially for ’370/C8 architecture:

- Register variables
- Register tracking/targeting
- Cost-based register allocation
- Near pointer arithmetic
- Tail merging

A.2.1 Register Variables

The compiler helps maximize the use of registers for storing local variables, parameters, and temporary values. Variables stored in registers can be accessed more efficiently than variables in memory. Register variables are particularly effective for pointers. See Example A–5, page A-3, and Example A–6, page A-11.

A.2.2 Register Tracking/Targeting

The compiler tracks the contents of registers to avoid reloading values if they are used again soon. Variables, constants, and structure references such as (a.b) are tracked through both straight-line code and forward branches. Register targeting also computes expressions directly into specific registers when required, as in the case of assigning to register variables or returning values from functions. See Example A–5, page A-3.

Example A–5. Register Variables and Register Tracking/Targeting

(a) C source

```
int gvar;
reg(int i, int j)
{
    ...
    gvar = call() & i & j;
    j    += gvar;
    ...
}
```

(b) TMS370/C8 8-bit C compiler output

```
_reg:
*
* R23       assigned to variable 'j'
* R24       assigned to variable 'i'
*
        ...
   CALL   _call  ; R8 = call()
   AND R24, R8   ; R8 &= i
   AND R23, R8   ; R8 &= j
   MOV R8, A
   MOV A, &_gvar ; gvar = R8
   ADD R8, R23   ; tracks gvar in R8, targets result into
                 ; R23 (j)
   ...
```

The compiler allocates parameters i and j into registers R23 and R24, as indicated by the comments in the assembly listing. Allocating j to R23 and tracking gvar in R8 allows the op-assignment j += gvar to be computed with a single instruction, targeting the result directly into j in R23.

A.2.3 Cost-Based Register Allocation

The optimizer, when enabled, allocates registers to user variables and compiler temporary values according to their type, use, and frequency. Variables used within loops are weighted to have priority over others, and those variables whose uses don't overlap may be allocated to the same register.

Example A–6. Strength Reduction, Induction Variable Elimination, Register Variables, and Loop Test Replacement

(a) C source

```
char a[10], b[10];
scale(int k)
{
   int i;

   for (i=0; i<10; ++i)
       a[i] = b[i] * k;
   ...
}
```

(b) TMS370/C8 C compiler output:

```
_scale:
*
* R10/11     assigned to temp var 'U$10'
* R12/13     assigned to temp var 'U$5'
* R14        assigned to temp var 'L$1'
* R15        assigned to variable 'k'
*
       MOV    *–2[FP], A   ; load scale parameter
       MOV    A, R15    ; move scale to R15
       MOVW   #_a, R11  ; setup pointer to A array
       MOVW   #_b, R13  ; setup pointer to B array
       MOV    #0ah, R14 ; load counter register
L2:
       MOV    *R13, A   ; load a[i]
       MPY    A, R15    ; a[i] * scale
       MOV    B, A
       MOV    A, *R11   ; b[i] = result
       INCW   #1, R13   ; point to next a[i]
       INCW   #1, R11   ; point to next b[i]
       DJNZ   R14, L2   ; decrement counter and repeat if
                        ; not done
```

Induction variable elimination and loop test replacement allow the compiler to recognize the loop as a simple counting loop and then generate a DNJZ instruction. Strength reduction turns the array references into efficient pointer references with autoincrements.

A.2.4 Near Pointer Arithmetic

For the TMS370/C8 8-bit C compiler, pointers are two bytes in size. Therefore, addition involving pointers requires 16-bit addition. Example A–7 illustrates pointer arithmetic:

Example A–7. Pointer Arithmetic

(a) C source

```
far char ary [];

char next (int i)
{
   return (ary [i]);
}
```

(b) TMS370/C8 C compiler output

```
_next:
         ADD       #LO(_ary+0),R3
         ADC       #HI(_ary+0),R2
         MOV       *R3,A
         MOV       A,R2
         RTS

   .global      _ary
   .bss      _ary,1
```

Notice that the index variable i is sign extended to 2 bytes and then is added to the base of ary, using 16-bit arithmetic.

Suppose that the character array ary is stored in the register file. Since the register file is limited to 256 bytes, the size of ary is limited to 256 bytes as well; using 16-bit addition with ary is not necessary, even though the pointers are 16 bits. Only the least significant eight bits are required. The TMS370/C8 8-bit C compiler detects instances of near pointer arithmetic (arithmetic involving a pointer that points to data in the register file) and only performs 8-bit arithmetic. See Example A–8.

Example A–8. Pointer Arithmetic Involving a Pointer That Points to Data in the Register File

(a) C source

```
near char ary [];

char next (int i)
{
   return (ary [i]);
}
```

(b) TMS370/C8 C compiler output

```
_next:
       MOVW   #_ary, R7
       MOV    *–2[FP], A
       ADD    A, R7
       MOV    *R7, A
       MOV    A, R8
```

If you intend to have a pointer pointing to data in the register file extend to data just outside the register file (rather than wrapping around to the beginning of the register file), then near pointer arithmetic optimization must be disabled.

Appendix B

Invoking the Compiler Tools Individually

The ’370/C8 C compiler offers you the option of invoking all of the tools at once by using the shell; however, you may find it necessary to invoke each tool individually. This appendix describes how you can invoke the compiler tools and the interlist utility individually.

For information about invoking the compiler tools in one step, see the description of how to use the shell program in Section 2.1, *Using the Shell Program to Compile, Assemble, and Link* (page 2-2).

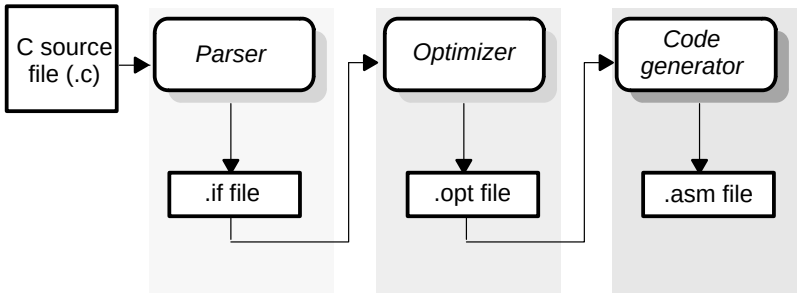
Topic **Page**

B.1 Which ’370/C8 Tools Can Be Invoked Individually?

To satisfy a variety of applications, you can invoke the compiler, the assembler, and the linker as individual programs.

- The **compiler** is made up of three distinct programs: the parser, the optimizer, and the code generator, which you can also invoke individually.

Figure B–1. Compiler Overview



The input for the **parser** is a C source file or, in program mode, all of the C source files that comprise an entire program. 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 on page B-4 describes how to invoke the parser.

The **optimizer** is an optional pass that runs between the parser and the code generator. The input is the intermediate file (.if) produced by the parser. When you run the optimizer, you choose the level of optimization. The optimizer performs the optimizations on the intermediate file and produces a highly efficient version of the file in the same intermediate file format. Section 2.4 on page 2-24 describes the optimizer.

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 on page B-9 describes how to invoke the code generator.

- 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 *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide.*

- The input for the **linker** is the COFF object file produced by the assembler. The linker also produces COFF object files, which may be executable. Chapter 3 describes how to run the linker. The linker is described fully in *Linker Description* of the *TMS370 and TMS370C8 8-Bit Microcontroller Family Assembly Language Tools User's Guide.*

- The inputs for the **interlist utilit**y are the assembly file produced by the compiler and the C source file. The utility produces an expanded assembly source file containing statements from the C file as assembly language comments. Section 2.6 on page 2-40 describes how the interlist utility works and Section B.5 on page B-11 describes how to invoke it.

B.2 Invoking the Parser

The first step in compiling a '370/C8 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. To invoke the parser, enter the following:

ac370 *input file* [*output file*] [*options*]

ac370 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, their functions, and a reference to the corresponding shell option. For a complete description of a parser option, refer to the page reference of the corresponding shell option.

Note: Using Wildcards

You cannot use wildcards if you invoke the tools individually.

Table B–1.Parser Options

Parser option	Function	Refer to	
		Shell option	Page
–c8	enable generation of TMS370C8 code	–c8	2-14
–d*name* [=*def*]	predefine macro name	–d*name* [=*def*]	2-14
–e	treat code-E errors as warnings	–pe	2-43
–f	generate function prototype listing file	–pf	2-23
–i*directory*	define #include search path	–i*directory*	2-14, 2-21
–k	allow K&R compatibility	–pk	4-21
–l (lowercase L)	generate .pp file†	–pl	2-22
–n	suppress #line directives	–pn	2-22
–o	preprocess only	–po	2-22
–q	suppress progress messages (quiet)	–q	2-15
–tp	relax pointer combination	–tp	2-17
–tf	relax prototype checking	–tf	2-17
–u*name*	undefine macro name	–u*name*	2-16
–w0	disable all warning messages	–pw0	2-43
–w1	enable serious warning messages (default)	–pw1	2-43
–w2	enable all warning messages	–pw2	2-43
–x0	disable inline expansion	–x0	2-35
–x1	enable inline expansion of intrinsic operators (default)	–x1	2-35
–x	enable inlining of user functions (implies –o2)	–x2	2-35
–?	enable trigraph expansion	–p?	2-23

† When running ac370 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.3 Invoking the Optimizer

The second step in compiling a ’370/C8 C program — optimizing — is optional. After parsing a C source file, you can choose to process the intermediate file with the optimizer. The optimizer attempts to improve the execution speed and reduce the size of C programs. If you invoke the optimizer individually (as described here), it optimizes for for speed over space. 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:

opt370 [*input file* [*output file*]] [*options*]

opt370 is the command that invokes the optimizer.

input file names the intermediate file produced by the parser. The optimizer assumes that the extension is .if.

output file names the intermediate file that the optimizer creates. If you don’t supply a filename for the output file, the optimizer uses the input filename with an extension of .opt.

options affect the way the optimizer processes the input file. The options that you use in standalone optimization are the same as those used for the shell. Table B–2 shows the optimizer options, their functions, and a reference to the corresponding shell option. For a complete description of an optimizer option, refer to the page reference of the corresponding shell option.

Note: Using Wildcards

You cannot use wildcards if you invoke the tools individually.

Table B–2.Optimizer Options

		Refer to	
Optimizer option	**Function**	**Shell option**	**Page**
–a	assume variables are aliased	–ma	2-15
–h0	inform the optimizer that your file alters a standard library function	–ol0	2-31
–h1	inform the optimizer that your file defines a standard library function	–ol1	2-31
–h2	inform the optimizer that your file does not define or alter library functions	–ol2	2-31
–i*size*	set automatic inlining size (–o3 only)	–oi*size*	2-35
–k	allow K&R compatibility	–pk	4-21
–n0	disable optimizer information file	–on0	2-31
–n1	produce optimizer information file	–on1	2-31
–n2	produce verbose optimizer information file	–on2	2-31
–mu	prevent optimizing unsigned 8-bit loop induction variables	–mu	2-15
–o0	optimize registers	–o0	2-24
–o1	use –o0 optimizations and optimize locally	–o1	2-24
–o2	use –o1 optimizations and optimize globally	–o2 or –o	2-25
–o3	use –o2 optimizations and optimize file	–o3	2-25

Table B–2.Optimizer Options (Continued)

		Refer to	
Optimizer option	**Function**	**Shell option**	**Page**
–p0	callable functions and modifiable variables are used in this module	–op0	2-29
–p1	no callable functions are used in this module but modifiable variables are used	–op1	2-29
–p2	no callable functions or modifiable variables are used in this module (default)	–op2	2-29
–p3	callable functions are used in this module but no modifiable variables are used	–op3	2-29
–x	expanded inline functions	–x*n*	2-35
–q	suppress progress messages (quiet)	–q	2-15
–s	interlist C source	–s	2-40
–z	optimize for space over speed. By default, opt370 optimizes for speed.	N/A	N/A

B.4 Invoking the Code Generator

The next step in compiling a ’370/C8 C program is to invoke the C code generator. As Figure B–1 on page B-2 shows, the code generator converts the intermediate file produced by the parser 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:

cg370 [*input file* [*output file* [*auxfile*]]] [*options*]

cg370 is the command that invokes the code generator.

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

output file names the assembly language source file that the code generator creates. If you don’t supply a filename for the output file, the code generator uses the input filename with the extension of .asm.

auxfile generates an auxiliary information file that you can refer to for information about stack size and function calls. The default filename is the C source filename with an .aux extension.

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. Table B–3 shows the code generator options, their functions, and a reference to the corresponding shell option. For a complete description of a code generator option, refer to the page reference of the corresponding shell option.

Note: Using Wildcards

You cannot use wildcards if you invoke the tools individually.

Table B–3.Code Generator Options

Code generator option	Function	Refer to	
		Shell option	Page
–a	assume variables are aliased	–ma	2-15
–mc	disable the default generation of automatic overlay directives	–mc	2-15
–mi	disable generation of .icalls directives when an indirect call is encountered	–mi	2-15
–o	enable C source level debugging	–g	2-14
–q	suppress progress messages (quiet)	–q	2-15
–z	retain the input file†	N/A	N/A

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

B.5 Invoking the Interlist Utility

Note: Interlisting With the Shell Program and the Optimizer

You can create an interlisted file by invoking the shell program with the –s or –ss option. If you use the –s shell option and invoke the optimizer, the optimizer, not the interlist utility, performs the interlist function. If you use the –ss shell option and invoke the optimizer, the interlist utility and the optimizer perform the interlist function.

The fourth step in compiling a ’370/C8 C program is optional; after you have compiled a program, you can run the interlist utility as an individual 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 output 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 a bracket { or }).
- **–q** removes banner and status information.
- **–r** removes symbolic debugging directives.

Note: Using Wildcards

You cannot use wildcards if you invoke the tools individually.

The interlist utility uses .line directives, produced by the code generator, to associate assembly code with C source. For this reason, you *must* use the –g shell 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:

```
cl370 –gk –mn function
```

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

To produce an interlist file, enter:

```
clist –r function
```

This creates an interlist file and removes the symbolic debugging directives from the file. The output from this example is function.cl.

Appendix C

Glossary

A

ANSI: American National Standards Institute.

absolute lister: A debugging tool that allows you to create assembler listings that contain absolute addresses.

aliasing: Aliasing occurs when a single object can be accessed in more than one way, such as when two pointers point to a single object. It can disrupt optimization, because any indirect reference could potentially refer to any other object.

allocation: A process in which the linker calculates the final memory addresses of output sections.

archive library: A collection of individual files that have been grouped into a single file.

archiver: A software program that allows you to collect several individual files into a single file called an archive library. The archiver also allows you to delete, extract, or replace members of the archive library, as well as to add new members.

assembler: A software program that creates a machine-language program from a source file that contains assembly language instructions, directives, and macro 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.

B

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: A sequence of eight adjacent bits operated upon as a unit.

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 .opt file from the optimizer and produces an assembly language source file.

command file: A file that contains linker or hex conversion utility options and names input files for the linker or hex conversion utility.

comment: A source statement (or portion of a source statement) that 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*.

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.

direct call: A function call where one function calls another function by giving the name of the called function.

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 emulates TMS370/C8 8-bit operation.

entry point: The starting execution point in target memory.

executable module: An object file that has been linked and can be executed in a target system.

expression: A constant, a symbol, or a series of constants and symbols separated by arithmetic operators.

external symbol: A symbol that is used in the current program module but defined in a different program module.

F

far variable: A variable that is allocated in RAM other than the TMS370/C8 register file, allowing for larger data sizes.

function inlining: Code for a function is inserted at the point of call. This saves the overhead of a function call, and allows the optimizer to optimize the function in the context of the surrounding code.

G

global: A kind of symbol that is either defined in the current module and accessed in another or accessed in the current module but defined in another.

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 actual code or data.

I

indirect call: A function call where one function calls another function by giving the address of the called function.

initialized section: A COFF section that contains executable code or initialized data. An initialized section can be built with the .data, .text, or .sect directive.

integrated preprocessor: The C preprocessor is integrated with the parser, allowing for faster compilation. Standalone preprocessing or preprocessed listing is also available.

interlist utility: This utility interlists your original C source files with the assembly language output from the assembler.

K

K&R C: Kernighan and Ritchie C, the de facto standard as defined in the first edition of *The C Programming Language* (K & R). Most K & R C programs written for earlier non-ANSI C compilers 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. The only assembler statement that can begin in column 1.

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 SPC.

loader: A device that loads an executable module into system memory.

M

macro: A user-defined routine that can be used as an instruction.

macro call: The process of invoking a macro.

macro definition: A block of source statements that define the name and the code that make up a macro.

macro expansion: The source statements that are substituted for the macro call and are subsequently assembled.

map file: An output file, created by the linker, that shows the memory configuration, section composition, and section allocation, as well as symbols and the addresses at which they were defined.

memory map: A map of target system memory space, which is partitioned into functional blocks.

N

near variable: A variable that is allocated in the on-chip register file of the TMS370/C8, allowing quick access.

O

object file: A file that has been assembled or linked and contains machine-language object code.

object library: An archive library made up of individual object files.

operand: The arguments, or parameters, of an assembly language instruction, assembler directive, or macro directive.

optimizer: A software tool that improves the execution speed and reduces the size of C programs.

options: Command parameters that allow you to request additional or specific functions when you invoke a software tool.

output module: A linked, executable object file that can be downloaded and executed on a target system.

output section: A final, allocated section in a linked, executable module.

P

parser: A software tool that reads the source file, performs preprocessing functions, checks the syntax, and produces an intermediate file that can be used as input for the optimizer or code generator.

preprocessor: A software tool that handles macro definitions and expansions, included files, conditional compilation, and preprocessor directives.

R

RAM model: An autoinitialization model used by the linker when linking C code. The linker uses this model when you invoke the linker with the –cr option. The RAM model allows variables to be initialized at load time instead of runtime.

relocation: A process in which the linker adjusts all the references to a symbol when the symbol's address changes.

ROM model: An autoinitialization model used by the linker when linking C code. The linker uses this model when you invoke the linker with the –c option. In the ROM model, the linker loads the .cinit section of data tables into memory, and variables are initialized at runtime.

runtime environment: Memory and register conventions, stack organization, function call conventions, and system initialization; also information on interfacing assembly language to C programs.

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 ultimately 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.

shell program: A utility that lets you compile, assemble, and optionally link in one step. The shell runs one or more source modules through: the compiler (including the parser, optimizer, and code generator), the assembler, and the linker.

source file: A file that contains C code or assembly language code that will be compiled or assembled to form an object file.

standalone preprocessor: Preprocessor expands macros, #include files, and conditional compilation. Integrated preprocessing, which includes parsing of instructions, is also available.

static: A kind of variable whose scope is confined to a function or a program. The values of static variables are not discarded when the function or program is exited; their previous value is resumed when the function or program is re-entered.

storage class: Any entry in the symbol table that indicates how a symbol should be accessed.

structure: A collection of one or more variables grouped together under a single name.

symbol: A string of alphanumeric characters that represents an address or a value.

symbol table: A portion of a COFF object file that contains information about the symbols that are defined and used by the file.

symbolic debugging: The ability of a software tool to retain symbolic information so that it can be used by a debugging tool such as a simulator or an emulator.

T

target system: The system on which the developed object code will be executed.

.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

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: A variable that may hold (at different times) objects of different types and sizes.

unsigned: A kind of value that is treated as a nonnegative number, regardless of its actual sign.

V

variable: A symbol representing a quantity that may assume any of a set of values.

veneer: A sequence of instructions that serves as an alternate entry point into a routine if a state change is required.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z