Reference Guide

TMS320C30 C Compiler

1989

Texas Instruments

TMS320C30 C Compiler

Reference Guide

1989

Digital Signal Processing Products

TMS320C30 C Compiler Reference Guide

Texas Instruments

IMPORTANT NOTICE

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

TI warrants performance of its semiconductor products to current specifications 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. Unless mandated by government requirements, specific testing of all parameters of each device is not necessarily performed.

TI assumes no liability for TI applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that 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.

TRADEMARKS

PC-DOS is a trademark of International Business Machines.

VAX and VMS are trademarks of Digital Equipment Corporation.

UNIX is a trademark of American Telephone and Telegraph.

SPOX is a trademark of Spectron Microsystems, Incorporated.

Preface

Read This First

This preface summarizes the chapters, lists related documentation, and describes the style and symbol conventions used in this book.

How to Use This Manual

This document contains the following chapters:

Chapter 1 **Introduction and Installation**
Provides an overview of the TMS320C30 software development tools, a walkthrough, and installation information.

Chapter 2 **C Compiler Operation**
Describes how to operate the C compiler and the CL30 program. Contains instructions for invoking CL30, which compiles, assembles, and links a C source file, and for invoking the individual compiler components. Discusses the interlist utility, filename specifications, compiler options, and using the linker and archiver with the compiler.

Chapter 3 **TMS320C30 C Language**
Discusses the differences between the C language supported by the TMS320C30 C compiler and standard Kernighan and Ritchie C language.

Chapter 4 **Runtime Environment**
Contains technical information on how the compiler uses the TMS320C30 architecture; discusses memory and register conventions, stack organization, function-call conventions, system initialization, and TMS320C30 C compiler optimizations; provides information needed for interfacing assembly language to C programs.

Chapter 5 **Runtime-Support Functions**
Describes the header files that are included with the C compiler, as well as the macros, functions, and types that they declare, summarizes the runtime-support functions according to category (header), and provides an alphabetical reference of the runtime-support functions.

Appendix A Compiler Error Messages
Provides the format of compiler error messages and lists all the fatal error messages.

Appendix B Preprocessor Directives
Describes the standard preprocessor directives that the compiler supports.

Appendix C Increasing Code Generation Efficiency
Presents guidelines for writing C programs that take advantage of the TMS320C30 C compiler optimizations.

Related Documentation

You should obtain a copy of ***The C Programming Language*** (first edition), by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey, 1978, to use with this manual.

You may find these two books useful as well:

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

Advanced C: Techniques and Applications Sobelman, Gerald E. and David E. Krekelberg. Que Corporation, 1985.

The following books, which describe the TMS320C30 and related support tools, are available from Texas Instruments:

- The ***Third-Generation TMS320 User's Guide*** (literature number SPRU031) discusses hardware aspects of the TMS320 family third-generation devices, including the TMS320C30. Topics in this user's guide include pin functions, architecture, stack operation, and interfaces; the manual also includes the TMS320C30 assembly language instruction set.
- The ***TMS320C30 Assembly Language Tools User's Guide*** (literature number SPRU035) describes the assembly language tools (assembler, linker, archiver, and code conversion utility), assembler directives, macros, common object file format, and symbolic debugging directives.

Style and Symbol Conventions

This document uses the following conventions:

- Program listings, program examples, interactive displays, filenames, and symbol names are shown in a special font. Examples use a bold version of the special font for emphasis. Here is a sample program segment:

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

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

  lnk30 *filenames*

  lnk30 is a command. This command has one parameter, indicated by *filenames*. When you use lnk30, the first parameter must be a filename.

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

  cl30 *[options] [filenames]*

  cl30 is a command. This command has two optional parameters, indicated by *options* and *filenames*. When you use cl30, no parameters are necessary; however, if you do indicate parameters, they should appear in this order.

Contents

Figures

Tables

Chapter 1

Introduction and Installation

The TMS320C30 is a high-performance CMOS floating-point microprocessor, optimized for digital signal processing applications. The TMS320C30 is a member of the third generation of TMS320 family digital signal processors..

The TMS320C30 is fully supported by a complete set of hardware and software development tools, including a C compiler, an assembler, linker, and archiver, a software simulator, and a full-speed emulator. Section1.1 describes these tools.

This reference guide describes the TMS320C30 C compiler. Its main purpose is to present the details and characteristics of this particular C compiler; it assumes that you already know how to write C programs. We suggest that you obtain a copy of *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie (published by Prentice-Hall); use this reference guide as a supplement to the Kernighan and Ritchie book.

Texas Instruments provides a hotline to assist you with technical questions about the TMS320 family products and development tools. The phone number is 713-274-2320.

The TMS320C30 C compiler can be installed on the following systems:

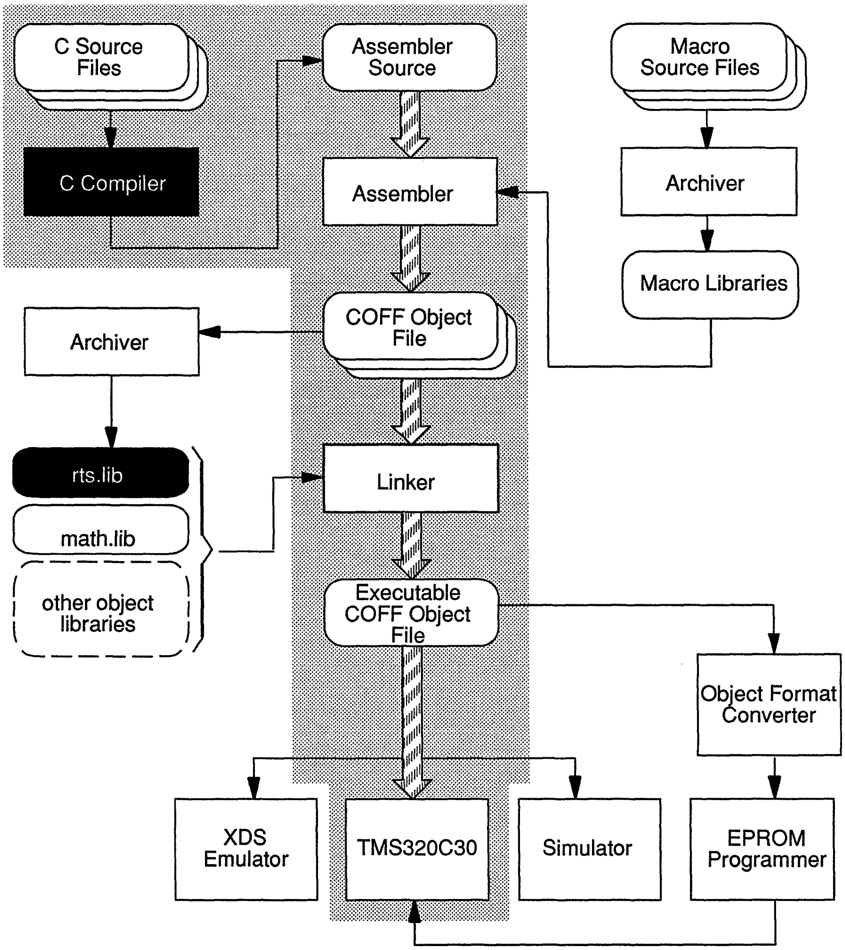
- IBM-PC/PC-DOS and compatibles
- VAX/VMS
- VAX/ULTRIX
- Workstations with UNIX
- Macintosh with MPW

Topics in this Chapter include:

1.1 Software Development Tools Overview

Figure 1–1 illustrates the TMS320C30 software development flow. The shaded portion of the figure highlights the typical software development path; the other portions are optional.

Figure 1–1. TMS320C30 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 TMS320C30 assembly language source code. A **CL30** program and an **interlist utility** are included in the compiler package.The CL30 program enables you to automatically compile, assemble, and link source modules. The interlist utility interweaves C source statements with assembly language output. Chapter 2 describes compiler, CL30, and interlist invocation and operation.
- The **assembler** translates assembly language source files into machine language object files.
- The **archiver** allows you to collect a group of files into a single archive file. (An archive file is also 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 to build a library of object modules.

  One object library, **rts.lib**, is shipped with the C compiler. This library contains standard runtime-support functions, compiler utility functions, and math functions that can be called in C programs. You can also create your own object libraries. To use an object library, you must specify the library name as linker input; the linker will include the members in the library that define the functions you call in a C program.
- 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.
- The main purpose of this development process is to produce a module that can be executed in a **TMS320C30 target system**. You can use one of several debugging tools to refine and correct your code before downloading it to a TMS320C30 system. These debugging tools share a common screen-oriented interface that displays and maintains machine status information and controls execution of the system that is being developed. Note that only *linked* object files can be executed.
  - The **simulator** is a software program that simulates TMS320C30 functions. The simulator can execute linked COFF object modules.
  - The **XDS emulator** is a PC-resident, realtime, in-circuit emulator with the same screen-oriented interface as the software simulator.
- An **object format converter** is also available; it converts a COFF object file into an Intel word, extended Tektronix hex, or TI-tagged object format file that can be downloaded to an EPROM programmer.

A software platform is also available for augmenting your TMS320C30 C compiler:

- **SPOX**

  SPOX is a high-level software interface designed specifically for digital signal processing and control applications. It is a system of software components that you can combine according to your needs. SPOX provides the common operating system functions of memory management, I/O, and multi-tasking. SPOX differs from traditional operating systems by supplying an optimized math and DSP library as well as real-time stream I/O. SPOX is available from Spectron Microsystems, Incorporated.

1.2 TMS320C30 C Compiler Overview

The TMS320C30 C compiler is a full-feature optimizing compiler that translates standard Kernighan and Ritchie C programs into TMS320C30 assembly language source. The following list describes key characteristics of the compiler:

- **Standard Kernighan and Ritchie C with Extensions**

  The compiler compiles standard C programs as defined by Kernighan and Ritchie's ***The C Programming Language*** (first edition). The compiler supports these standard extensions: enumeration types, structure assignments, passing structures to functions, and returning structures from functions. A future release of the compiler will support the full ANSI standard. For more information, refer to Chapter 3.

- **32-Bit Data Sizes**

  All data sizes (char, short, int, long, float, and double) are 32 bits. This allows all types of data to take full advantage of the TMS320C30's 32-bit integer and floating-point arithmetic capabilities. For more information, refer to Section 3.2 on page 3-4.

- **Big and Small Memory Models**

  The compiler supports two memory models.The small memory model enables the compiler to efficiently access memory by restricting the global data space to a single 64K-word data page. The big memory model allows unlimited space. For more information, refer to Section 4.1 on page 4-2.

- **Optimization**

  The compiler uses several advanced techniques for generating efficient, compact code from C source. For more information the C compiler's optimization techniques, refer to Section 4.9 on page 4-28 and Appendix C.

- **Assembly Source Output**

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

- **COFF Object Files**

  The COFF format allows you to define you system's memory map at link time. This maximizes performance by enabling you to link C code and data objects into specific memory areas. COFF also provides rich support for source-level debugging.

- **ROM-able Code**

  For stand-alone embedded applications, the compiler enables you to link all code and initialization data into ROM.

- **ANSI Standard Runtime Support**

  The compiler package comes 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, exponential, and hyperbolic functions. Functions for I/O and signal handling are not included because these are target-system specific. For more information, refer to Chapter 5.

- **Flexible Assembly Language Interface**

  The compiler has straight-forward calling conventions, allowing you to easily write assembly and C functions that call each other. For more information, refer to Chapter 4.

- **CL30 Compiler Shell Program**

  The compiler package includes a CL30 shell program which enables you to compile, assemble, and link programs in a single step. For more information, refer to Chapter 2.

- **Source Interlist Utility**

  The compiler package includes a utility that interlists your original C source statements into the assembly language output of the compiler. This utility provides you with an easy method for inspecting the assembly code generated for each C statement. For more information, refer to Section 2.7 on page 2-14.

1.3 Getting Started

The TMS320C30 C compiler has three parts: a preprocessor, a parser, and a code generator. The compiler produces a single assembly language source file that must be assembled and linked. The simplest way to compile, assemble, and link a C program is to use the **CL30** program which is included with the compiler. This section provides a quick walkthrough so that you can get started without reading the entire reference guide.

1) Create a sample file called `function.c` that contains the following code:

```
/************************************/
/*          function.c              */
/* (Sample file for walkthrough)    */
/************************************/
#include "stdlib.h"

int abs(i)
   int i;
{
   register int temp = i;
   if (temp < 0) temp = -temp;
   return (temp);
}
```

2) Invoke CL30 to run the compiler and assembler.

   `cl30 function` ↵

   CL30 prints the following information as it compiles the program:

```
[function]
C Pre-Processor,                Version 2.00
(c) Copyright 1987, 1989, Texas Instruments Inc.
TMS320C30 C Compiler,           Version 2.00
(c) Copyright 1987, 1989, Texas Instruments Inc.
   "function.c" ==> abs
TMS320C30 C Codegen,            Version 2.00
(c) Copyright 1987, 1989, Texas Instruments Inc.
   "function.c" ==> abs
TMS320C30 COFF Assembler,       Version 2.00
(c) Copyright 1987, 1989, Texas Instruments Inc.
 PASS 1
 PASS 2

No Errors, No Warnings
```

   CL30 runs the three compiler passes and the assembler as follows:

   cpp30 → C Preprocessor
   cc30 → C Parser
   cg30 → Code Generator
   asm30 → Assembler

By default, CL30 deletes the assembly language output file from the compiler after it's assembled. If you wish to inspect the assembly language output of the compiler, use the **–k** option on CL30.

3) Also by default, CL30 creates a COFF object file as output; however, if you use the **–z** option, the output will be an executable object module. The following examples walk you through the two ways of achieving an executable object module:

   a) The example above creates an object file called `function.obj`. To create an executable object module, link the object file with the runtime-support library `rts.lib`:

   ```
   lnk30  -c  function  -o function.out  -l rts.lib ↵
   ```

   This examples uses the –c linker option because the code came from a C program. The –l option tells the linker that the input file `rts.lib` is an object library. The –o option names the output module, `function.out`; if you don't use the –o option, the linker names the output module `a.out`.

   b) In this example, CL30 runs the linker directly by using the **–z** option, followed by the linker options.

   ```
   cl30  function  -z -o function.out  -l rts.lib ↵
   ```

   This example runs the three compiler passes, the assembler, and the linker as follows:

cpp30	→	C Preprocessor
cc30	→	C Parser
cg30	→	Code Generator
asm30	→	Assembler
lnk30	→	Linker

4) The TMS320C30 includes an **interlist utility**. This program interweaves the C source statements as comments in the assembly language compiler output, allowing you to inspect the assembly language generated for each line of C. To run the interlist utility, invoke CL30 with the **–s** option. For example:

   ```
   cl30  function  -z -s -o function.out  -l rts.lib ↵
   ```

   The output of the interlist utility is written to the assembly language file created by the compiler. (The CL30 **–s** option implies **–k**; that is, when you use the interlist utility, the assembly file is automatically retained.)

For more information about invoking the C compiler, the interlist utility and the CL30 program, refer to Chapter 2.

1.4 Compiler Installation

This section contains step-by-step instructions for installing the TMS320C30 C compiler. Refer to the following sections for installation information:

1.4.1 Installing the C Compiler on IBM-PCs with PC-DOS

The C compiler package is shipped on double-sided, dual-density diskettes. The compiler executes in batch mode and requires 512K bytes of RAM.

These instructions are for both hard-disk systems and dual floppy drive systems (however, we recommend that you use the compiler on a hard-disk system). On a dual-drive system, the PC-DOS system diskette should be in drive B. The instructions use these symbols for drive names:

A: Floppy disk drive for hard disk systems; source drive for dual-drive systems.

B: Destination or system disk for dual-drive systems.

C: Winchester (hard disk) for hard-disk systems.

Follow these instructions to install the software:

1) Make backups of the product diskettes.
2) Create a directory to contain the C compiler. If you're using a dual-drive system, put the disk that will contain the tools into drive B.
   - On *hard-disk* systems, enter:
     `MD C:\C30TOOLS` ↵
   - On *dual-drive* systems, enter:
     `MD B:\C30TOOLS` ↵
3) Copy the C compiler package onto the hard disk or the system disk. Put the product diskette in drive A; if you're using a dual-drive system, put the disk that will contain the tools into drive B.
   - On *hard-disk* systems, enter:
     `COPY A:\*.* C:\C30TOOLS\*.*` ↵
   - On *dual-drive* systems, enter:
     `COPY A:\*.* B:\C30TOOLS\*.*` ↵
4) Repeat steps 1 through 3 for each product diskette.

1.4.2 Installing the C Compiler on VAX/VMS

The TMS320C30 C compiler tape was created with the VMS BACKUP utility at 1600 BPI. These tools were developed on version 4.5 of VMS. If you are using an earlier version of VMS, you may need to relink the object files; refer to the release notes for relinking instructions.

Follow these instructions to install the compiler:

1) Mount the tape on your tape drive.

2) Execute the following VMS commands. Note that you must create a destination directory to contain the package; in this example, `DEST:directory` represents that directory. Replace `TAPE` with the name of the tape drive you are using.

```
$  allocate             TAPE:
$  mount/for/den=1600   TAPE:
$  backup               TAPE:c30.bck/select=[master.c30c...] DEST:[directory...]
$  dismount             TAPE:
$  dealloc              TAPE:
```

3) The product tape contains a file called `setup.com`. This file sets up VMS symbols that allow you to execute the tools in the same manner as other VMS commands. Enter the following command to execute the file:

   `$ @setup DEST:directory` ↵

   This sets up symbols that you can use to call the various tools. As the file is executed, it will display the defined symbols on the screen.

1.4.3 Installing the C Compiler on Workstations with UNIX

The TMS320C30 C compiler product tape was made using the tar utility. Follow these instructions to install the compiler:

1) Mount the tape on your tape drive.

2) Make sure that the directory you store the tools in is the current directory.

3) Enter the tar command for your system; for example,

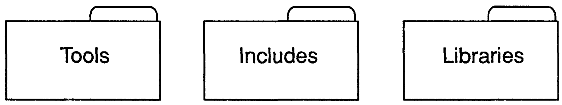
   `tar  x` ↵

   This copies the entire tape into the directory. The tar command varies from system to system; consult your system documentation for proper use of the tar command.

1.4.4 Installing the C Compiler on Macintosh with MPW

The TMS320C30 compiler package runs **only** under the Macintosh Programmer's Workshop (MPW). MPW is a complete software development environment for Macintosh Computers that can be purchased through for Apple. These tools cannot be run on a Macintosh without MPW.

The C compiler is shipped on a double-sided, 800k, 3 1/2" diskette. The disk contains three folders:



Use the Finder to display the disk contents and copy the files into your MPW environment:

1) The *Tools* directory contains all the programs and the batch files for running the compiler. Copy this directory in with your other MPW tools (MPW tools are usually in the folder {MPW}Tools.)

2) The *Includes* directory contains the header files (`.h` files) for the runtime-support functions. Many of these files have names that conflict with commonly-used MPW header files, so you should keep these header files separate from the MPW files. Copy the contents of the *Includes* directory into a new folder, and use the C_DIR environment variable. For information describing how to create a path to this folder, refer to Section 2.8.1.1 on page 2-18.

3) The *Libraries* folder contains the compiler's runtime-support object and source libraries. You can copy these files into the folder that you created for the header files, or you can copy them into a new folder. If you copy them into a new folder, use the C_DIR environment variable to create a path to this folder as well.

Chapter 2

C Compiler Operation

The TMS320C30 C compiler is made up of three programs: the preprocessor, the parser, and the code generator. After compiling a program, you must assemble and link it with the TMS320C30 assembler and linker. The CL30 program, included with the compiler, enables you to automatically compile, assemble, and link one or more source modules.

The compiler package also includes a utility that interlists your original C source statements into the assembly language output of the compiler, enabling you to inspect the assembly code generated for each C statement. The interlist utility is explained in Section 2.7.

If you choose to run the three compiler steps individually, Section 2.8 describes how to run the preprocessor, parser, and code generator individually.

Topics in this chapter include:

2.1 C Compiler Overview

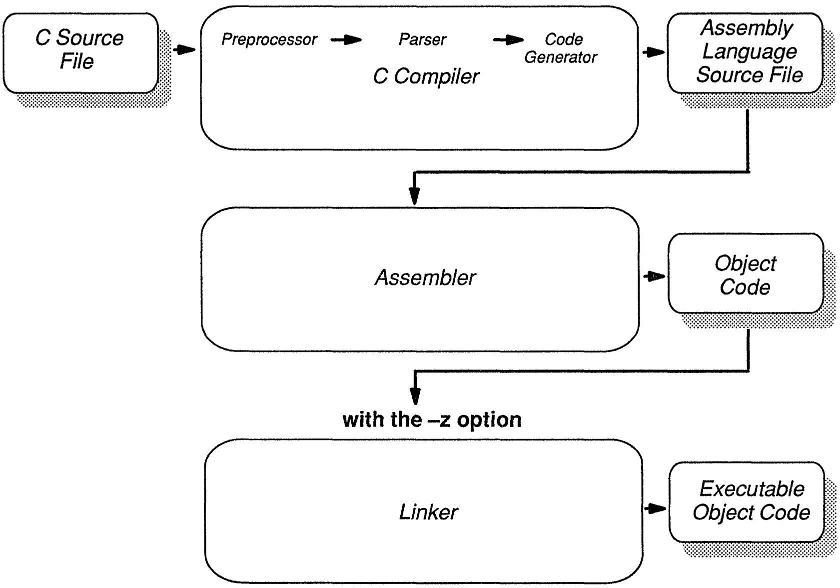
The TMS320C30 C compiler is made up the preprocessor, the parser, and the code generator.

After you have compiled a program, you must assemble and link it with the TMS320C30 assembler and linker. A program called **CL30** is provided with the compiler which automatically runs one or more source modules through:

- the three compiler passes,
- the assembler, and
- if the –z option is used, the linker.

Figure 2–1 illustrates CL30 with and without the –z option.You can invoke CL30 with compiler, assembler, and linker options, and CL30 will automatically vector the options to the appropriate program. You may also set CL30 default options by using the C_OPTION environment variable; these defaults options are used every time you run CL30.

Figure 2–1. CL30 Overview



2.2 Invoking the C Compiler

To run the compiler, enter

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

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

Options and filenames can be specified in any order on the command line, but if you use the **–z** option, it must follow all filenames and compiler options.

2.3 Filename Specifications

The input files specified on the command line can be C source files, assembly source files, or object files. CL30 uses filename extensions to determine the file type.

Extension	File Type	File Description
.c	C source	compiled, assembled, and (linked)
.asm	assembly source	assembled and (linked)
.s* (extension begins with s)	assembly source	assembled and (linked)
.o* (extension begins with o)	object file	linked
none (**.c** assumed)	C source	compiled, assembled, and (linked)

Extensions and filenames are not case sensitive. Files without extensions are assumed to be C source files and a .c extension is appended. You can override these file type interpretations by using the **–f** option as follows:

–fa *file* for an assembly file

–fc *file* for a C source file

–fo *file* for an object file

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

You can compile and assemble source files with a single command. Here are some examples.

1) To compile all the files in a directory, enter:

```
cl30 *.c ↵
```

2) To compile a source file named `hilev.c` and two assembly files called `lowlev.asm` and `lowlev2.asm`, enter:

```
cl30 hilev lowlev1.asm lowlev2.asm ↵
```

As CL30 encounters each source file, it prints the filename in square brackets [for c files] or angle brackets <for asm files>. Progress information is output from each of the compiler passes unless the **–q** option is specified. If you use the –q option, only the source filenames print. If you use the **–qq** option, no progress information prints except error messages. For example, the output from compiling a single module might be:

```
$ cl30 symtab
[symtab]
C Pre-Processor                              Version 2.00
(c) Copyright 1987, 1989, Texas Instruments Incorporated
TMS320C30 C Compiler                         Version 2.00
(c) Copyright 1987, 1989, Texas Instruments Incorporated
    "symtab.c":==>  main
    "symtab.c":==>  lookup
TMS320C30 C Codegen                          Version 2.00
(c) Copyright 1987, 1989, Texas Instruments Incorporated
    "symtab.c":==>  main
    "symtab.c":==>  lookup
TMS320C30 COFF Assembler                     Version 2.00
(c) Copyright 1987, 1989, Texas Instruments Incorporated
    PASS 1
    PASS 2

No Errors, No Warnings
```

Using the quiet option (–q) to compile multiple files, you might get:

```
$ cl30 -q symtab file seek.asm
[symtab]
[file]
<seek.asm>
```

2.4 Options

Command line options control the operation of both CL30 and the programs it calls.

2.4.1 Option Conventions

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, *–mrb* is equivalent to *–mr* and *–mb*. Options that have parameters, such as *–d*, must be specified separately.

Table 2–1 summarizes the following options: general, preprocessor, assembler, runtime model, filename, linker, and environment variable options. Section 2.4.2 provides an in-depth description of each of these options.

Table 2–1. Options Summary Table

General Options Usage: cl30 [–options] filenames. . . [–z link_options. . .]			
–c	no linking (negates –z)	–q	quiet
–d*name*	predefine *name*	–qq	super quiet
–g	symbolic debugging	–s	C source interlist
–i<*dir*>	#include search path	–u*name*	undefine *name*
–k	keep .asm file	–z	link, options follow
–n	compile only		

Preprocessor Options –p<options. . . >			
–pc	preprocess only	–pp	no #line directive

Assembler Options –a<options. . . >			
–al	assembly listing file	–ax	cross-reference file
–as	keep labels as symbols	–ap	preprocess first

Runtime Model Options	
–ma	assumes aliased variables
–mb	enables the big memory model
–mm	enables the short multiply
–mn	normal optimization, even with debug

Table 2–1. Options Summary Table (Continued)

Runtime Model Options *(continued)*	
–mr	lists register use information
–mv	volatile variables
–mx	avoids TMX silicon bugs

–f options (File Specifiers)	
–fa *file*	assembly language file (default for .asm or .s*)
–fc *file*	C source file (default for .c or no ext)
–fo *file*	object file (default for .o*)

Linker Options (all options following –z go to the linker)			
–a	absolute output	–ar	relocatable output
–c	ROM initialization	–cr	RAM initialization
–e *sym*	entry point	–f *val*	fill value
–h	global symbols static	–i *dir*	library search path
–l *lib*	library name	–m *file*	map filename
–o *file*	output filename	–r	relocatable output
–s	strip symbol table	–u *sym*	undefine *sym*

Environment Variables	
setenv C_OPTION "options"	to set default options
setenv C_DIR "dirs"	to set cpp and linker search paths

2.4.2 Option Descriptions

This section contains descriptions of general, compiler, preprocessor, assembler, runtime model, and linker options.

- **General Options**

  –c suppresses the linking option; it causes CL30 to not run the linker even if –z is specified. This option is especially useful when you have –z specified in the C_OPTION environment variable and you don't want to link. For more information, refer to Section 2.6 on page 2-13.

–g causes the compiler to generate symbolic directives for use with a high-level language debugger.

–i*dir* adds *dir* to the list of directories to be searched for #include files. You can use this option multiple times to define several directories; be sure to separate *–i* options with spaces. Note that if you don't specify a directory name, the preprocessor ignores the –i option.

–k keeps the .asm file. Normally, CL30 deletes the output assembly language file after assembly is finished, but using –k allows you to retain the assembly source output from the compiler.

–n causes CL30 to compile only. If you use –n, the specified source files are compiled but not assembled or linked. This option overrides –z and –c. The output of –n is assembly source 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 interweaves C source statements into the assembly language output of the compiler, allowing you to inspect the code generated for each C statement. This option implies that the –k option is specified. For more information about the interlist utility, refer to Section 2.7 on page 2-14.

–z enables the linking option; it causes CL30 to run the linker on specified object files. –z must follow all source files and compiler options on the command line. All arguments that follow –z on the command line are passed to and interpreted by the linker.

–f –f options override default interpretations for source file extensions. If your naming conventions do not conform to those of CL30, you can use –f options to specify exactly which files are C source files, assembly files, or object files. You can insert an optional space between the –f option and the filename.

–fa*file*	This file is an assembly source file.
–fc*file*	This file is C source file.
–fo*file*	This file is an object file.

If you have a C source file called `cfile.s` and an assembly file called `assy`, use –f to force the correct interpretation:

```
cl30 -fc cfile.s -fa assy
```

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

- **Compiler Options**

–d*name*[=def]	pre-defines *name* for the preprocessor. This is equivalent to inserting *#define name def* at the top of each C source file. If the optional *[def]* if omitted, –d*name*[=def] sets *name* equal to 1.
–u*name*	undefines the predefined constant *name*.

- **Preprocessor Options**

–pc causes the compiler to preprocess only. –pc runs the preprocessor on the specified source files and retains the comments. The remaining compiler passes, the assembler, and the linker are not run.

–pp suppresses line and file information. –pp causes the preprocessor to suppress its normal location directives of the form:

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

–pp is sometimes useful when compiling machine-generated code.

- **Assembler Options**

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

–ap enables preprocessing. –ap runs the C preprocessor on the assembly source before assembling them.

–as retains labels. Label definitions are written to the COFF symbol table for use with symbolic debugging.

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

For more information about assembler options, see Section 4.2, page 4-3 in the *TMS320C30 Assembly Language Tools User's Guide*.

- **Runtime Model Options**

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

–mb selects the big memory model. –mb allows unlimited space for global data, static data, and constants. In the small memory model, which is the default, this space is limited to 64k words. For more information, refer to Section 4.1 on page 4-2.

–mm enables the short multiply. –mm generates MPYI instructions for integer multiples rather than runtime-support calls. If your application does not need 32x32-bit integer multiplication, use –mm to enable the MPYI instruction because it is significantly faster (but it performs only 24x24-bit multiplication). For more information, refer to Section 4.8 on page 4-26.

–mn normal optimazation, even with debug. When you generate symbolic debugging information with the –g switch, the code generator disables certain optimizations that inhibit debugging. You can use –mn to re-enable these optimizations and generate exactly the same code as without –g.

–mr lists register use information. After the code generator compiles each C statement, –mr lists register contents tables as comments in the assembly file. –mr is useful for inspecting code that is difficult to follow due to register tracking optimizations.

–mv assumes variables are volatile. Disables register tracking optimizations. Variables are always read from memory each time they are accessed.

–mx avoids early silicon bugs. –mx enables the code generator to work around some of the known hardware bugs in early TMX320C30 devices.

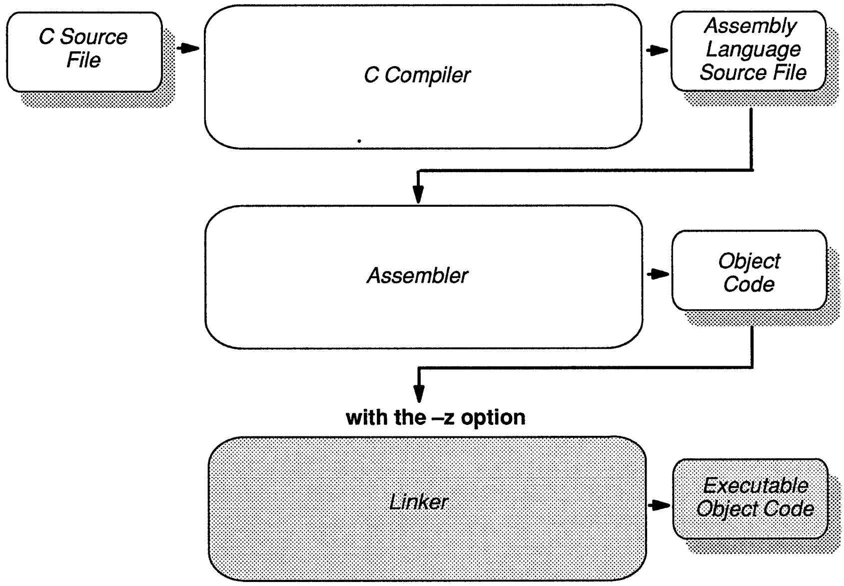
- **Linker Options**

  All command line input following –z is passed to the linker. Table 2–1, on page 2-6, summarizes the linker options. For more information about linker options, see Section 9.3, page 9-4, in the *TMS320C30 Assembly Language Tools User's Guide*.

2.5 Running the Linker with CL30

CL30, by default, does not run the linker; however, you can enable the linker by using the **–z** option.

Figure 2–2. CL30 Overview with the Linker



2.5.1 –z CL30 option

When using –z to enable linking, remember:

- –c suppresses –z, so do not use –c if you want linking enabled,
- –z must follow all source files and compiler options on the command line, and
- –z divides the command line into compiler options (before –z) and linker options (following –z)

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.

The order in which the linker processes arguments can be important, especially for command files and libraries. When you use CL30 to run the linker, it passes arguments to the linker in the following order.

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

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

```
cl30 -sq -mm  *.c -z c.cmd -o prog.out -l rts.lib
```

First, *cl30* compiles all the files with **.c* extensions using the *–sq* and *–mm* options. Second, because ***–z*** is specified, the linker runs the resulting object files using the the linker command file *c.cmd*, the *–o* option to name the output file, and the *–l* option to include the runtime-support library.

For more information about linker operation, refer to Section 2.9 on page 2-24 in this manual and Chapter 9, Linker Description, in the *TMS320C30 Assembly Language Tools User's Guide*. For more information about linker options, refer to Section 9.3 in the *TMS320C30 Assembly Language Tools User's Guide*.

2.5.2 –c CL30 Option

Passing the **–c** option to CL30 overrides –z and disables linking. This option is helpful when you have specified –z in the C_OPTION environment variable and want to selectively disable linking with –c on the command line.

2.5.3 –c and –cr Linker Options

The **–c linker option** has a different function than, and is independent of, the **–c CL30 option**. By default, CL30 automatically uses the –c option that tells the linker to use C source linking conventions (ROM model of initialization). If you want to use –cr (RAM model of initialization) rather than –c, you can pass –cr as a linker option.

2.6 Using the C_OPTION Environment Variable

You can set up default options for CL30 using the C_OPTION environment variable. After CL30 reads the entire command line, it reads the C_OPTION environment variable and processes it.

Options in the environment variable are specified in the same way and have the same meaning as they do on the command line.

For example, if you want to always run quietly, enable symbolic debugging, and link, then set up the C_OPTION environment variable as follows:

Host	Enter:
DOS	set C_OPTION=–qg –z
UNIX	setenv C_OPTION "–qg –z"
VAX/VMS	assign "–qg –z" C_OPTION
MPW	set C_OPTION "–qg –z"; export C_OPTION

Using the –z option in the environment variable enables linking. In the examples above, each time you run CL30, it will run the linker. Any options following –z on the command line are passed to the linker; likewise, any options following –z on the options line are passed to the linker. This enables you to use the environment variable to specify default compiler and linker options and then specify additional compiler and linker options on the CL30 command line. If you have set –z in the environment variable and want to compile (or assemble) only, use the –c option of CL30. These additional examples assume C_OPTION is set as shown above:

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

2.7 Interlist Utility Operation

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

2.7.1 Invoking the Interlist Utility Using the –s CL30 Option

The easiest way to invoke the interlist utility is to use the –s CL30 option. To compile and run the interlist utility on a program called `function.c`, enter:

```
cl30 -s function
```

The interlist runs a separate pass between the code generator and the assembler. It reads both the assembly and C source files, merges them, and writes the C statements into the assembly file as comments (beginning with >>>>). The output assembly file is assembled normally. The –s option automatically prevents CL30 from deleting the interlisted assembly language file.

Figure 2–3 shows a typical interlisted assembly file.

Figure 2–3. An Example of an Interlisted Assembly File

```
;>>>>              main()
;>>>>                 int i, j;
*********************************************
*          FUNCTION DEF : _main             *
*********************************************
_main:
        PUSH       FP
        LDI        SP, FP
        ADDI       2, SP
;>>>>                    i += j;
        LDI        *+FP(1),R3
        ADDI       *+FP(2),R3
        STI        R3,*+FP(1)
;>>>>                    j = i + 123;
        ADDI       123,R3
        STI        R3,*+FP(2)
        SUBI       2,SP
        RETS
```

2.7.2 Invoking the Interlist Utility Outside CL30

Even if you are not using CL30, you can still use the interlist utility. After you have compiled a program, you can run the interlist utility as a standalone program from the command line. To run the interlist utility from the command line, the syntax is:

clist *asmfile [outfile] [–options]*

clist is the command that invokes the interlist utility.

asmfile is the assembly language output from the compiler.

outfile names the interlisted output file. If you omit this, the file has the same name as the assembly file with the the extension *.cl*.

options control the operation of the utility as follows:

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

The interlist utility uses the .line directives produced by the code generator to associate assembly code with C source. For this reason, you **must** 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** option to remove them from the interlisted file.

The following example shows how to compile and interlist `function.c`.

Function	To invoke, enter:	Comments
compile	cl30 –gk function	compile, use debug, keep assembly
interlist	clist –r function	interlist, remove debug

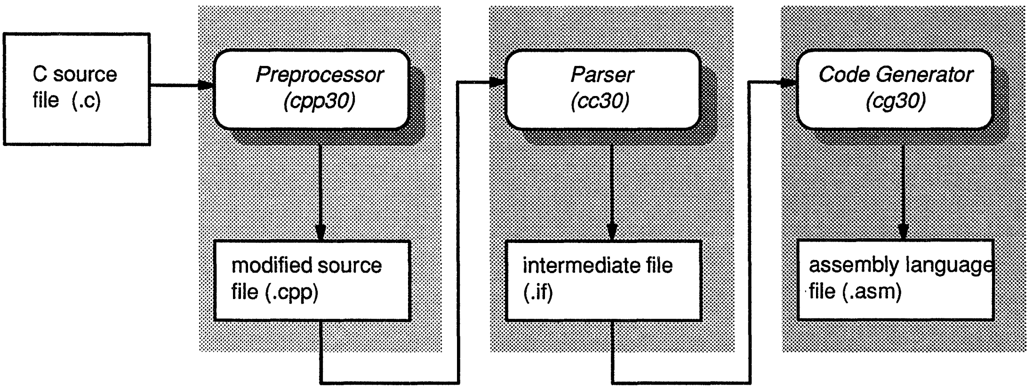
The output from this example is `function.cl`.

2.8 Operating the Preprocessor, the Parser, and the Code Generator Individually

The TMS320C30 C compiler is made up of three distinct programs: the preprocessor, the parser, and the code generator. This section provides information about how to run the individual programs.

- The input for the **preprocessor** is a C source file (as described in Kernighan and Ritchie). The preprocessor produces a modified version of the source file. Section 2.8.1 describes how to run the preprocessor.
- The input for the **parser** is the modified source file produced by the preprocessor. The parser produces an intermediate file. Section 2.8.2 describes how to run the parser.
- The input for the **code generator** is the intermediate file produced by the parser. The code generator produces an assembly language source file. Section 2.8.3 describes how to run the code generator.

Figure 2–4. Compiling a C Program



Refer to the following sections for more information:

2.8.1 Preprocessing C Code

The first step in compiling a TMS320C30 C program is to invoke the C preprocessor. The preprocessor handles macro definitions and substitutions, #include files, line number directives, and conditional compilation. As Figure 2–4 shows, the preprocessor uses a C source file as input, and produces a modified source file that can be used as input for the C parser.

To invoke the preprocessor as a standalone program, enter:

cpp30 *[input file [output file]] [options]*

cpp30 is the command that invokes the preprocessor.

input file names a C source file that the preprocessor uses as input. If you don't supply an extension, the preprocessor assumes that the extension is ***.c***. If you don't specify an input file, the preprocessor will prompt you for one.

output file names the modified source file that the preprocessor creates. If you don't supply a filename for the output file, the preprocessor uses the input filename with an extension of ***.cpp***.

options affect the way the preprocessor processes your input file. Options *are not* case sensitive. Valid options include:

- **–c** copies comments to the output file. If you don't use this option, the preprocessor strips comments.
- **–d*name*[=def]** See Compiler Options, page 2-9.
- **–i*dir*** adds `dir` to the list of directories to be searched for #include files. See Compiler Options, page 2-9.
- **–p** suppresses line number and file information.
- **–q** suppresses the banner and status information.

This preprocessor is described in Kernighan and Ritchie; additional information can be found in that book. The preprocessor supports the same preprocessor directives that are summarized in Appendix B of that book. All preprocessor directives begin with the character **#**, which must appear in column 1 of the source statement. Any number of blanks and tabs may appear between the # sign and the directive name.

The C preprocessor maintains and recognizes five predefined macro names:

`__LINE__`	represents the current line number (maintained as a decimal integer).
`__FILE__`	represents the current filename (maintained as a C string).
`__DATE__`	represents the date that the module was compiled (maintained as a C string).
`__TIME__`	represents the time when this module was compiled (maintained as a C string).
`_320C30`	identifies the compiler as the TMS320C30 C compiler; this symbol is defined as the constant *1*.

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

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

would translate into a line such as:

```
printf(%s %s", "May 1 1989", "13:58:17");
```

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

2.8.1.1 Specifying Alternate Directories for Include Files

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

#include "*filename*" **or** **#include** <*filename*>

The *filename* names an include file that the preprocessor reads statements from; you can enclose the *filename* in double quotes or in angle brackets. The *filename* can be a complete pathname or a filename with no path information.

- If you provide path information for *filename*, the preprocessor uses that path and *does not look* for the file in any other directories.
- If you do not provide path information and you enclose the *filename* in **double quotes**, the preprocessor searches for the file in this order:
  1) The directory that contains the current source file. (The current source file refers to the file that is being processed when the preprocessor encounters the #include directive.)
  2) Any directories named with the –i preprocessor option.
  3) Any directories set with the environment variable C_DIR.

- If you do not provide path information and you enclose the *filename* in **angle brackets**, the preprocessor searches for the file in:
  1) Any directories named with the –i preprocessor option.
  2) Any directories set with the environment variable C_DIR.

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

You can augment the preprocessor's directory search algorithm by using the –i preprocessor option or the environment variable C_DIR.

2.8.1.2 –i Preprocessor Option

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

cpp30 –i *pathname*

You can use up to 10 –i options per invocation; each –i option names one *pathname*. In C source, you can use the #include directive without specifying any path information for the file; instead, you can specify the path information with the –i option. For example, assume that a file called `source.c` is in the current directory; `source.c` contains the following directive statement:

```
#include  "alt.c"
```

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

	Pathname for `alt.c`	Invocation Command
DOS	`c:\C30\files\alt.c`	`cpp30 -ic:\C30\files source.c`
VMS	`[C30.files]alt.c`	`cpp30 -i[C30.files] source.c`
UNIX	`/C30/files/alt.c`	`cpp30 -i/C30/files source.c`
MPW	`:C30:files:alt.c`	`cpp30 -i:C30 :files source.c`

Note that the include filename is enclosed in double quotes. The preprocessor first searches for `alt.c` in the current directory, because `source.c` is in the current directory. Then, the preprocessor searches the directory named with the –i option.

2.8.1.3 Environment Variable

An environment variable is a system symbol that you define and assign a string to. The preprocessor uses an environment variable named **C_DIR** to name alternate directories that contain include files. The commands for assigning the environment variable are:

DOS:	`set`	`C_DIR=` *pathname;another pathname ...*
VMS:	`assign`	*"pathname;another pathname ... "* `C_DIR`
UNIX:	`setenv`	`C_DIR` *"pathname;another pathname ... "*
MPW:	`set` `export`	`C_DIR` *"pathname;another : pathname ..."* `C_DIR`

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

For example, assume that a file called `source.c` contains these statements:

```
#include  <alt1.c>
#include  <alt2.c>
```

The table below lists the complete pathnames for these files and shows how to invoke the preprocessor; select the row for your operating system.

	Pathname for `alt1.c` `and alt2.c`	**Invocation Command**
DOS	`c:\C30\files\alt1.c` `c:\sys\alt2.c`	`set C_DIR=c:\sys c:\exec\files` `cpp30 -ic:\C30\files source.c`
VMS	`[C30.files]alt1.c` `[sys]alt2.c`	`assign C_DIR "[sys] [exec.files]"` `cpp30 -i[C30.files] source.c`
UNIX	`/C30/files/alt1.c` `/ygs/alt2.c`	`setenv C_DIR "/sys /exec/files` `cpp30 -i\C30\files source.c`

MPW	`:C30 :files :alt1 .c`
	`:sys:alt2.c`
		`set C_DIR " :sys :files "`
		`export C_DIR`
		`cpp30 -i:C30 :files source.c`

Note that the include filenames are enclosed in angle brackets. The preprocessor first searches for these files in the directories named with C_DIR and finds `alt2.c`. Then, the preprocessor searches in the directories named with the –i option and finds `alt1.c`.

The environment variable remains set until you reboot the system or reset the variable by entering:

DOS:	`set`	`C_DIR=`
VMS:	`deassign`	`C_DIR`
UNIX:	`setenv`	`C_DIR " "`
MPW:	`unset`	`C_DIR`

2.8.2 Parsing C Code

The second step in compiling a TMS320C30 C program is to invoke the C parser. The parser reads the modified source file produced by the preprocessor, parses the file, checks the syntax, and produces an intermediate file that can be used as input for the code generator. (Note that the input file can also be a C source file that has not been preprocessed.)

To invoke the parser as a standalone program, enter:

cc30 *[input file [output file]] [options]*

cc30 is the command that invokes the parser.

input file names the preprocessed C source file that the parser uses as input. If you don't supply an extension, the parser assumes that the extension is ***.cpp***. If you don't specify an input file, the parser will prompt 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 the way the parser processes the input file. Valid options include:

- **–q** suppresses the banner and status information.
- **–z** tells the parser to retain the input file (the intermediate file created by the preprocessor). If you don't specify –z, the parser deletes the `.cpp` input file. (The parser **never** deletes files with the .c extension.)

Most errors are fatal; that is, they prevent the parser from generating an intermediate file and must be corrected before you can finish compiling a program. Some errors, however, merely produce warnings that hint of problems but do not prevent the parser from producing an intermediate file.

When the parser encounters function definitions, it prints a progress message that contains the name of the source file and the name of the function. Here is an example of a progress message:

"filename.**c***"*: **=> main**

This type of message shows how far the compiler has progressed in its execution and helps you to identify the locations of an error. You can use the –q option to suppress these messages.

If the input file has an extension of `.cpp`, the parser deletes it upon completion *unless* you use the –z option. If the input file has an extension other than `.cpp`, the parser does not delete it.

The intermediate file is a binary file; do not try to inspect or modify it in any way.

2.8.3 Generating Assembly Language Code

The third step in compiling a TMS320C30 C program is to invoke the C code generator. As Figure 2–4 on page 2-16 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 TMS320C30 assembler. The code generator produces re-entrant relocatable code, which, after assembling and linking, can be stored in ROM.

To invoke the code generator as standalone, enter:

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

cg30 is the command that invokes the code generator.

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

output file names the assembly language source file that the code generator creates. If you don't supply a filename for the output file, the code generator uses the input filename with an extension of ***.asm***.

tempfile names a temporary file that the code generator creates and uses. The default filename for the temporary file is the input filename appended with an extension of ***.tmp***. The code generator deletes this file after using it.

options affect the way the code generator processes the input file. Valid options include:

- **–a** assumes variables are aliased. For more information, refer to Section 2.4 on page 2-9
- **–b** tells the compiler to generate code for the big memory model.
- **–m** enables the short multiply. For more information, refer to Section 2.4 on page 2-9.

–n normal optimization, even with debug. When you generate symbolic debugging information with the –g switch, the code generator disables certain optimizations that inhibit debugging. You can use –mn to re-enable these optimizations and generate exactly the same code as without –g.

–o tells the code generator to place symbolic debugging directives in the output file. See Appendix B of the *TMS320C30 Assembly Language Tools User's Guide* for more information about these directives.

–q suppresses the banner and status information.

–v assumes variables are volatile. Variables are always read from memory each time they are accessed. For more information, refer to Section 2.4 on page 2-9

–x avoids early silicon bugs. –x enables the code generator to work around some of the known hardware bugs in early TMX320C30 devices.

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

2.9 Linking a C Program

The TMS320C30 C compiler and assembly language tools support modular programming by allowing you to compile and assemble individual modules and then link them together. To link compiled and assembled code, enter:

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

lnk30 is the command that invokes the linker.

–c/–cr are options that tell the linker to use special conventions that are defined by the C environment. Note that when you use CL30 to link, CL30 passes –c to the linker automatically.

filenames are object files created by compiling and assembling C programs.

–o name.out names the output file. If you don't use the –o option, the linker creates an output file with the default name of `a.out`.

rts.lib `rts.lib` is an archive library that contains C runtime-support functions. (The –l option tells the linker that a file is an object library.) The library is shipped with the C compiler. If you're linking C code, you must use `rts.lib`. Whenever you specify a library as linker input, the linker includes and links only those library members that resolve undefined references.

For example, you can link a C program consisting of modules `prog1`, `prog2`, and `prog3` (the output file is named `prog.out`):

```
lnk30 -c prog1 prog2 prog3 -l rts.lib -o prog.out ⏎
```

The linker uses a default allocation algorithm to allocate your program into memory. You can use the MEMORY and SECTIONS linker directives to customize the allocation process.

2.9.1 Runtime Initialization and Runtime Support

All C programs must be linked with the `boot.obj` object module; this module contains code for the C boot routine. The `boot.obj` module is a member of the runtime-support object library, `rts.lib`. To use the module, simply use –c or –cr and include the library in the link:

```
lnk30  -c  -l  rts.lib ...
```

The linker automatically extracts `boot.obj` and links it in when you use the –c or –cr option.

When a C program begins running, it must execute `boot.obj` first. 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 generate a branch to `_c_int00` so that the TMS320C30 executes `boot.obj` first. The `boot.obj` module contains code and data for initializing the runtime environment; the module performs the following tasks:

- Sets up the system stack.
- Processes the runtime initialization table and autoinitializes global variables (in the ROM model).
- Disables interrupts and calls `_main`.
- Calls `exit` when `main` returns.

Chapter 5 describes additional runtime-support functions that are included in `rts.lib`. If your program uses any of these functions, you must link `rts.lib` with your object files.

2.9.2 Sample Linker Command File

Figure 2–5 shows a typical linker command file that can be used to link a C program. The command file in this example is named `link.cmd`.

Figure 2–5. An Example of a Linker Command File

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

-c                 /* ROM autoinitialization model       */
-m example.map     /* Create a map file                  */
-o example.out     /* Output file name                   */
main.obj           /* First C module                     */
sub.obj            /* Second C module                    */
asm.obj            /* Assembly language module           */
-l rts.lib         /* Runtime-support library            */
-l matrix.lib      /* Object library                     */
```

- The command file first lists several linker options:

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

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

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

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

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

- Finally, the command file lists all the object libraries that the linker must search. (The libraries are specified with the –l linker option.) Because this is a C program, the runtime-support library `rts.lib` **must** be included. This program uses several routines from an archive library called `matrix.lib`, so it is also named as linker input. Note that only the library members that resolve undefined references are linked in.

To link the program using this command file, simply enter:

```
lnk30  link.cmd ↵
```

This example uses the default memory allocation described in Chapter 9 of the *TMS320C30 Assembly Language Tools User's Guide.* If you want to specify different MEMORY and SECTIONS definitions, refer to that user's guide.

2.9.3 Autoinitialization (RAM and ROM Models)

The C compiler produces tables of data for autoinitializing global variables. Section 4.10.1.1, page 4-31, 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.* A loader copies the initialization data into the variables in the .bss section; thus, no runtime initialization is performed at boot time. This enhances performance by reducing boot time and saving memory used by the initialization tables.

  For more information about the RAM model, refer to Section 4.10.1.2 on page 4-32.

- **ROM Model** (–c linker option)

  Global variables are initialized at *run time*. The .cinit section is loaded into memory along with all the other sections. The linker defines a special symbol called `cinit` that points to the beginning of the tables in memory. When the program begins running, the C boot routine copies data from the tables 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.

  For more information about the ROM model, refer to Section 4.10.1.3 on page 4-33.

2.9.4 The –c and –cr Linker Options

The following list outlines what happens when you invoke the linker with the –c or –cr option.

- 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 `rts.lib`.
- The .cinit output section is padded with a termination record so that the loader (RAM model) or the boot routine (ROM model) knows when to stop reading the initialization tables.
- In the RAM model (–cr option):
  - The linker sets the symbol `cinit` to –1. This indicates that the initialization tables are not in memory, so no initialization is performed at run time.
  - The STYP_COPY flag (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.
- 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.

2.10 Using the Archiver with C

An archive file (or library) is a partitioned file that contains complete files as members. The TMS320C30 archiver is a software utility that allows you to collect files into a single archive file. The archiver also allows you to manipulate a library by adding members to it or by extracting, deleting, or replacing members. The *TMS320C30 Assembly Language Tools User's Guide* contains complete instructions for using the archiver.

After compiling and assembling multiple files, you can use the archiver to collect the object files into a library. You can specify an archive file as linker input. The linker is able to discern which files in a library resolve external references, and it links in only those library members that it needs. This is useful for creating a library of related functions; the linker links in only the functions that a program calls. The library `rts.lib` is an example of an object library.

You can also use the archiver to collect C source programs into a library. The C compiler cannot choose individual files from a library; you must extract them before compiling them. However, this can be useful for managing files and for transferring source files between systems. The library `rts.src` is an example of an archive file that contains source files.

For more information about the archiver, see the *TMS320C30 Assembly Language Tools User's Guide*.

Chapter 3

TMS320C30 C Language

The C language that the TMS320C30 C compiler supports is based on the Unix System V C language that is described by Kernighan and Ritchie, with several additions and enhancements to provide compatibility with ANSI C. The most significant differences are:

- The data type *enum* has been added.
- A member of a structure can have the same name as a member of another structure (unique names aren't required).
- Structures and unions can be passed as parameters to functions, returned from functions, and assigned directly.

This chapter compares the two forms of C language and presents only the *differences* between them. The TMS320C30 C compiler supports standard Kernighan and Ritchie C except as noted.

References to Kernighan and Ritchie's C Reference Manual (Appendix A of *The C Programming Language*) are used throughout this chapter.

Topics in this chapter include:

3.1 Identifiers, Keywords, and Constants

K&R 2.2 *Identifiers*

- In TMS320C30 C, the **first 31 characters of an identifier are significant** (in K&R C, 8 characters are significant). This also applies to external names.
- **Case is significant**; uppercase characters are different from lowercase characters in all TMS320C30 tools. This also applies to external names.

K&R 2.3 *Keywords*

TMS320C30 C reserves **three additional keywords**:

```
asm
void
enum
```

K&R 2.41 *Integer Constants*

- All integer constants are of type *int* (signed, 32 bits long) unless they have an *L* or *U* suffix. If the compiler encounters an invalid digit in a constant (such as an 8 or 9 in an octal constant), it issues a warning message.
- You can append a letter suffix to an integer constant to specify its type:
  - Use *U* as a suffix to declare an unsigned integer constant.
  - Use *L* as a suffix to declare a long integer constant.
  - Combine the suffixes to declare an unsigned long integer constant.

  Suffixes can be upper or lower case.
- Here are some examples of integer constants:

```
1234;            /* int                */
0xFFFFFFFFu;     /* unsigned int       */
0L;              /* long int           */
077613LU;        /* unsigned long int  */
```

K&R 2.43 *Character Constants*

In addition to the escape codes listed in K&R, the TMS320C30 C compiler **recognizes the escape code \v** in character and string constants as a vertical tab character (ASCII code 11).

Added Type – Enumeration Constants

An enumeration constant is **an additional type of integer constant** that is not described by K&R. An identifier that is declared as an enumerator can be used in the same manner that an integer constant can be used. (For more information about enumerators, refer to Section 3.5 on page 3-7.)

K&R 2.5 String Constants

- K&R C does not limit the length of string constants; *however*, TMS320C30 C **limits the length of string constants to 255 bytes**.
- Any characters that follow an embedded null byte within a string constant are ignored; in other words, the first null byte terminates a string.

  This does not apply to strings used to initialize arrays of characters.
- **Identical string constants are stored as a single string**, not as separate strings as in K&R C.

  This does not apply to strings used for autoinitialization of arrays of characters.

3.2 TMS320C30 C Data Types

K&R 4.0 *Added Type and Equivalent Types*

- The *char* data type is signed. A separate type, *unsigned char*, is also supported.
- *char*, *short*, *long*, and *int* are functionally equivalent types. Any of these types can be declared unsigned.
- The properties of *enum* types are identical to those of *unsigned int*.

K&R 4.0 *Added Types*

- **An additional type**, called *void*, can be used to declare a function that returns no value. The compiler checks that functions declared as *void* do not return values and that they are not used in expressions. Functions are the only type of objects that can be declared *void*.
- The compiler also supports a type that is a **pointer to void** (`void *`). An object of type void * can be converted to and from a pointer to an object of any other type without explicit conversions (casts). However, such a pointer cannot be used indirectly to access the object that it points to without a conversion. For example,

```
void   *p, *malloc();
char   *c;
int    i;
p  =  malloc();       /* Legal                     */
p  =  c;              /* Legal, no cast needed     */
p  =  &i;             /* Legal, no cast needed     */
c  =  malloc();       /* Legal, no cast needed     */
i  =  *p;             /*Illegal, dereferencing
                         void pointer              */
i  =  *(int *)p;      /* Legal, dereferencing
                         casted void pointer       */
```

K&R 4.0 *Derived Types*

TMS320C30 C allows any type declaration to have **up to six derived types.** Constructions such as *pointer to*, *array of*, and *function returning* can be combined and applied a maximum of six times.

For example:

```
int (* (*n[][]) () ) ();
```

translates as:

1) an array of
2) arrays of
3) pointers to
4) functions returning
5) pointers to
6) functions returning integers

It has six derived types, which is the maximum allowed.

Structures, unions, and enumerations are not considered derived types for the purposes of these limits.

An additional constraint is that the derived type cannot contain more than three array derivations. Note that each dimension in a multidimensional array is a separate array derivation; thus, arrays are limited to three dimensions in any type definition. However, types can be combined using typedefs to produce any dimensioned array.

For example, the following construction declares `x` as a four-dimensional array:

```
typedef int dim2[][];
dim2 x[][];
```

Table 3–1. Summary of TMS320C30 Data Types (K&R 2.6)

Type	Size
char	8 bits, signed ASCII
unsigned char	8 bits, ASCII
short	16 bits
unsigned short	16 bits
int	32 bits
unsigned int	32 bits
long	32 bits
unsigned long	32 bits
pointers	32 bits
float	32 bits Range: $\pm 5.88 \times 10^{(-39)}$ through $\pm 1.70 \times 10^{38}$
double	64 bits Range: $\pm 1.11 \times 10^{(-308)}$ through $\pm 8.99 \times 10^{308}$
enum	1—32 bits

3.3 Object Alignment

- All objects except bit fields are aligned on 32-bit (one word) boundaries. Bit fields are always unsigned and can be from 1 to 32 bits in length. Adjacent fields are packed into adjacent bits of a word, but they do not overlap words; if a field would overlap into the next word, the entire field is placed into the next word. (A bit field never crosses a word boundary.) Fields are packed as they are encountered; the least significant bits of a structure word are filled first.
- When the compiler allocates space for a structure, it allocates as many words as are needed to hold all of the structure's members. In an array of structures, each structure begins on a word boundary.

3.4 Expressions

Added type – Void Expressions

A function of type *void* has no value (returns no value) and cannot be called in any way except as a separate statement or as the left operand of the comma operator. Functions can be declared or typecast as *void*.

K&R 7.2 *Unary Operators in Expressions*

The value yielded by the *sizeof* operator is calculated as the total number of bits used to store the object divided by 32. (32 is the number of bits in a character.) *Sizeof* can be legally applied to bit fields. If the result is not an integer, it is rounded up to the nearest integer. For example,

```
sizeof(int) == sizeof(short) == sizeof(char) ==
sizeof(long) == sizeof(float) == sizeof(double) == 1
```

3.5 Declarations

K&R 8.1 Register Variables

- The TMS320C30 C compiler allows you to use up to eight register variables in a function:
  - Two TMS320C30 registers (R4 and R5) are reserved for the first two integer register variables in a function.
  - Two registers (R6 and R7) are reserved for float or double register variables.
  - Four registers (AR4—AR7) are reserved for pointer register variables.

  For more information about register variables, refer to Section 4.3, Register Conventions, on page 4-12.
- All integer types (signed or unsigned), floats, doubles, and pointers, can be declared as registers.

K&R 8.2 Type Specifiers in Declarations

In addition to the type specifiers listed in K&R, objects can be declared with *enum* specifiers.

TMS320C30 C allows **more type name combinations** than K&R C allows. The adjectives *long* and *short* can be used with or without the word *int*; the meaning is the same in either case. The word *unsigned* can be used in conjunction with any integer type or alone; if alone, *int* is implied. *Long float* is a synonym for *double*. Otherwise, only one type specifier is allowed in a declaration.

K&R 8.4 Passing/Returning Structures to/from Functions

Contrary to K&R, TMS320C30 C allows functions to return structures and unions.

Structures and unions can be used as parameters to functions, can be directly assigned, and can be returned from functions.

K&R 10 External Definitions

Formal parameters to a function can be declared as type *struct*, *union*, or *enum* (in addition to the normal function declarations) because TMS320C30 C allows these types of objects to be passed to functions.

K&R 8.5, K&R 14.1 Structure and Union Declarations

Bit fields are limited to a maximum size of 32 bits. Any integer type can be declared as a field. Fields are always treated as unsigned, regardless of definition.

K&R states that structure and union member names must be mutually distinct. In TMS320C30 C, **members of different structures or unions can have the same name.** However, this requires that references to the member be fully qualified through all levels of nesting.

TMS320C30 C allows assignment to and from structures, passing structures as parameters, and returning structures from functions.

K&R states that the compiler determines the type of structure reference by the member name. Because TMS320C30 C does not require member names to be unique, this statement does not apply. All structure references must be fully qualified as members of the structure or union in which they were declared.

Added Type – Enumeration Declarations

Enumerations allow the use of named integer constants in TMS320C30 C. The syntax of an enumeration declaration is similar to that of a structure or union. The keyword *enum* is substituted for *struct* or *union*, and a list of enumerators is substituted for the list of members.

Enumeration declarations have a *tag*, as do structure and union declarations. This tag can be used in future declarations without repeating the entire declaration.

The list of enumerators is simply a comma-separated list of identifiers. Each identifier can be followed by an equal sign and an integer constant. If no enumerator is followed by an = sign and a value, then the first enumerator is assigned the value 0, the next is 1, the next is 2, etc. An identifier with an assigned value assumes that value, and subsequent enumerators continue counting by one from there. The assigned value can be negative, but counting still continues by positive one.

Unlike structure and union members, enumerators share their name space with ordinary variables and, therefore, must not conflict with variables or other enumerators in the same scope.

Enumerators can appear wherever integer constants are required and, therefore, can be used in arithmetic expressions, case expressions, etc. In addition, explicit integer expressions can be assigned to variables of type *enum*. The compiler does no range checking to insure the value will fit in the enumeration field. The compiler does, however, issue a warning message if an enumerator of one type is assigned to a variable of another.

Here's an example of an enumeration declaration:

```
enum    color {
        red,
        blue,
        green = 10,
        orange,
        purple = -2,
        cyan }       x;
```

This statement declares x as a variable of type *enum*. The enumerators and their assigned values are:

```
red: 0
blue: 1
green: 10
orange: 11
purple: -2
cyan: -1
```

32 bits are allocated for the variable `x`. Legal operations on these enumerators include:

```
x = blue;
x = blue + red;
x = 100;
i =  red;              /* assume i is an int */
x = i + cyan;
```

3.6 Initialization of Static and Global Variables

K&R 8.6

An important difference between K&R C and TMS320C30 C is that **external and static variables are not preinitialized to zero** unless the program explicitly does so or unless it is specified by the linker.

If a program requires external and static variables to be preinitialized, you can use the linker to accomplish this. In the linker control file, use a fill value of 0 in the .bss section:

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

3.7 Lexical Scope Rules

K&R 11.1

The lexical scope rules stated in K&R apply to TMS320C30 C also, except that structures and unions each have distinct name spaces for their members. In addition, the name space of both enumeration variables and enumeration constants is the same as for ordinary variables.

3.8 asm Statement

Additional Statement

TMS320C30 C has another statement not mentioned in K&R: **the asm statement**. The compiler copies asm statements from the C source directly into the assembly language output file. The syntax of the asm statement is:

asm("*assembler text*");

The assembler text must be enclosed in double quotes. All the usual character string escape codes retain their definitions. The assembler text is copied directly to the assembler source file. Note that the assembler source statement **must** begin with a label, a blank, or a comment indicator (asterisk or semicolon).

Each asm statement injects one line of assembly language into the compiler output. A series of asm commands places the statements sequentially into the output with no intervening code.

Asm statements do not follow the syntactic restrictions of normal statements and can appear anywhere in the C source, even outside blocks. However, they are ignored when they appear in a list of declarations.

Note:

Be extremely careful not to disrupt the C environment with asm commands. 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. The asm command is provided so you can access features of the hardware, which by definition C is unable to access. Specifically, do not use this command to change the value of a C variable; however, you can use it safely to read the current value of a variable.

The asm command is very useful in the context of register variables. A register variable is a variable in a C program that is declared to reside in a machine register. The TMS320C30 C compiler allows up to 8 machine registers to be allocated to register variables. These 8 registers, combined with the asm command, provide a means of manipulating data independently of the C environment.

Chapter 4

Runtime Environment

This chapter describes the TMS320C30 C runtime environment. To ensure successful execution of C programs, it is critical that all runtime code maintain this environment. If you write assembly language functions that interface to C code, follow the guidelines in this section.

Topics in this chapter include:

4.1 Memory Model

The C compiler treats memory as a single linear block of memory that is partitioned into subblocks of code and data. Each block of code or data that a C program generates will be placed in its own contiguous space in memory. The compiler assumes that the full 24-bit address space is available in target memory.

Note that the **linker**, *not the compiler*, defines the memory map and allocates code and data into target memory. The compiler assumes nothing about the types of memory that are available, about any locations that are not available (holes), or about any locations that are reserved for I/O or control purposes. 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.

4.1.1 Sections

The compiler produces five relocatable blocks of code and data; these blocks, called **sections**, can be allocated into memory in a variety of ways to conform to a variety of system configurations. For more information about sections, read Chapter 3, Introduction to Common Object File Format, of the *TMS320C30 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 two initialized sections, .text and .cinit.
  - The **.text section** is an initialized section that contains all the executable code as well as string literals.
  - The **.cinit section** is an initialized section that contains tables for initializing variables and constants.
- **Uninitialized sections** reserve space in memory (usually in RAM). A program can use this space at run time for creating and storing variables. The C compiler creates three uninitialized sections, .bss, .stack, and .sysmem.
  - The **.bss section** is an uninitialized section. It reserves space for global and static variables, and in the small model (described in Section 4.1.2), it reserves space for tables of long immediate constants. 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 **.stack section** is an uninitialized section. It allocates memory for the system stack, which is used to pass arguments to functions and to allocate local variables.

- The **.sysmem section** is an uninitialized section. It allocates memory for use by the dynamic memory functions `malloc`, `calloc`, and `realloc`. If a C program does not use these functions, then the compiler does not create the .sysmem section.

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 with the same name to create six output sections. The complete program is made up of these five output sections, plus the assembler's .data section. You can place these output sections anywhere in the address space, as needed, to meet system requirements. The .text, .cinit, and .data sections are usually linked into either ROM or RAM. The .bss, .stack, and .sysmem sections should be linked into some type of RAM.

For more information about allocating sections into memory, refer to Chapter 9, Linker Description, in the *TMS320C30 Assembly Language Tools User's Guide.*

4.1.2 Big and Small Memory Models

The compiler supports two memory models that affect the treatment of the .bss section:

- The **small memory model**, which is the default model, requires the entire .bss section to fit in a single 64K memory page (65,536 words). This means that the total space for all static and global data in the program must be less than 64K and that the .bss section cannot span any 64K address boundaries. The compiler sets the Data Page Pointer register (DP) during runtime initialization to point to the beginning of .bss. Then, the compiler can access all objects in .bss (global and static variables, plus constant tables) with direct addressing (@symbol) without modifying the DP.

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

Here's an example of assembly language code that uses the LDP instruction to set up the DP register before accessing a global variable.

```
***    Assume that _x is a global variable    ***
       LDP   _x       ; 1 extra word, 1 cycle
       LDI   @_x,R0   ; 3 cycles (2 pipeline delays)
```

To use the big model, invoke the compiler with the –mb option; for more information, refer to Section 2.4 on page 2-6.

Neither model restricts the size of the .text or .cinit sections.

Both models restrict the size of a single function to 32K (32768 words of code) or less; this allows the compiler to generate relative conditional jumps over the entire range of a function.

Note:

Be sure all code in the system is compiled under the same model. *Mixed-model code will not run.* The runtime-support library that is provided with the compiler (`rts.lib`) is compiled with the **small model**. To use the library under the big model, you must:

1) Extract all the source files from the source archive `rts.src`.
2) Recompile these extracted files; be sure to invoke the code generator with the –b option.
3) Archive the object files into a new library.

Neither model restricts the size of the dynamic memory area in the .sysmem section because dynamically allocated objects are accessed with indirect, rather than direct, addressing. Thus, if you have large data objects, it is advantageous to allocate them dynamically rather than declare them as static or global variables; for more information, refer to Section 4.1.4 on page 4-6.

Under the small model, be careful when linking the .bss section; it must be less than 64K words and it cannot span any 64K page boundaries. Neither the compiler nor the linker checks for restrictions on .bss against the model used. If you choose to use the small model and your code does not conform to small-model restrictions, the code will not run. If you want to verify that the .bss section is fully contained within a 64K memory page, check the link map after linking.

4.1.3 C System Stack

The C compiler uses a stack to:

- Allocate local variables,
- Pass arguments to functions, **and**
- Save temporary results.

The compiler uses two registers to manage the stack:

SP is the **stack pointer**; it marks the top of the stack.

AR3 is the **frame pointer** (**FP**); it points to the beginning of the current local frame. (A local frame is an area on the stack that is used for storing arguments and local variables.) Each function invocation causes a new local frame to be created at the top of the stack.

The C environment automatically manipulates these registers when a C function is called. If you interface assembly language routines to C, be sure to use the registers in the same way that the C compiler uses them.

The C initialization module, `boot.asm`, allocates memory for the stack in an uninitialized, named section called **.stack**. This module also defines a constant named `STACK_SIZE` that determines the size of the stack. The default stack size is 400h (1K words); this size allows the stack to fit into one of the on-chip RAM blocks. You can change the amount of memory that is reserved for the stack by following these steps:

1) Extract `boot.asm` from the source library `rts.src`.
2) Edit `boot.asm`; change the value of the constant `STACK_SIZE` to the desired stack size.
3) Reassemble `boot.asm` and replace the resulting object file, `boot.obj`, in the object library `rts.lib`.
4) Replace the copy of `boot.asm` that's in `rts.src` with the new, edited version.

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

Note:

The compiler provides no means to check for stack overflow during compilation or at run time. If the stack overflows, your system will probably crash. Be sure that you allow enough space for the stack to grow; either set `STACK_SIZE` to an appropriate amount or allocate the .stack section last.

4.1.4 Dynamic Memory Allocation

The runtime-support library supplied with the compiler contains several functions (such as `malloc`, `calloc`, and `realloc`) that allow you to dynamically allocate memory for variables at run time. This is accomplished by declaring a large memory pool, or heap, and then using the functions to allocate memory from the heap. Dynamic allocation is not a standard part of the C language; it is provided by standard runtime-support functions.

An assembly language module called `sysmem.asm` defines this memory pool as an uninitialized, named section called **.sysmem**. The module also defines a constant named `__SYSMEM_SIZE` that determines the size of the memory pool; the default size is 800h (2K words). You can change the size of the memory pool by following these steps:

1) Extract `sysmem.asm` from the source library `rts.src`.
2) Edit `sysmem.asm`; change the value of the constant `__SYSMEM_SIZE` to the desired memory pool size.
3) Reassemble `sysmem.asm` and replace the resulting object file, `sysmem.obj`, in the object library `rts.lib`.
4) Replace the copy of `sysmem.asm` that's in `rts.src` with the new, edited version.

Dynamically allocated objects are not addressed directly (they are always accessed with pointers), and the memory pool is in a separate section; therefore, the dynamic memory pool can have an unlimited size, even in the small memory model. The size of the pool does not affect the 64K limit on global and static variables. This allows you to use the more efficient small memory model even if you declare large data objects. To conserve space in .bss, you can allocate large arrays from the heap instead of declaring them as global or static. For example, instead of a declaration such as:

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

use a pointer, and call the `malloc` function:

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

Note:

If you don't use dynamic allocation—that is, if you don't use `calloc`, `malloc`, and similar functions—then it is not necessary to allocate the .sysmem section at link time.

4.1.5 RAM and ROM Models

The C compiler produces code that is suitable for use as firmware in a ROM-based system. In such a system, the initialization tables in the .cinit section are stored in ROM. At system initialization time, the C boot routine copies data from these tables (in ROM) to the initialized variables in .bss (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 run time). You can specify this *to the linker* by using the –cr linker option.

For more information about autoinitialization, refer to Section 4.10 on page 4-30.

4.2 Object Representation

4.2.1 Storage of Data Types

- All basic types are 32-bits wide and stored in individual words of memory. No packing is performed, except for bit fields, which are packed into words. Bit fields are allocated from the LSB to the MSB in the order in which they are declared.
- No object has any type of alignment requirement; any object can be stored on any 32-bit word boundary. Objects that are members of structures or arrays are stored just as they are individually. Members are not packed into structures or arrays (unless the members are bit fields).
- The integral types *char*, *short*, *int*, and *long* are all equivalent, as are their unsigned counterparts. Objects of type *enum* are also represented in 32-bit words.
- The *float* and *double* types are equivalent; both types specify objects represented in the TMS320C30's 32-bit floating-point format.

4.2.2 Long Immediate Values

The TMS320C30 instruction set has no immediate operands that are longer than 16 bits. The compiler occasionally needs to use constants that are too long to be immediate operands. This occurs with signed integer constants that have more than 15 significant non-sign bits, with unsigned integers that have more than 16 significant bits, or with floating-point constants that have more than 11 significant non-sign bits in the mantissa. The compiler uses the .word and .float assembler directives to build a table in memory that contains all such constants. Constants in the table are then accessed like global variables, using direct addressing. Section 4.2.5, page 4-10, describes the structure of the constant table.

4.2.3 Addressing Global Variables

The compiler generates the addresses of global or static symbols for indexing arrays or manipulating pointers. Because these addresses may be up to 24 bits wide, and immediate operands are limited to 16 bits, these addresses are treated like long constants as described in Section 4.2.2. The compiler generates addresses into the constant table using the .word assembler directive. Section 4.2.5, page 4-10, describes the structure of the constant table.

4.2.4 Character String Constants

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

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

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

  When a string is used as an initializer, it is simply treated as an initialized array; each character is a separate initializer. For more information about autoinitialization, refer to Section 4.10 on page 4-30.

- It can be used as a pointer; for example:

  ```
  printf("abc");
  ```

  When a string is used as a pointer, the string itself is defined in the .text section using the .byte assembler directive, along with a unique label that points to the string; the terminating 0 byte is included. For example, the following line defines the string `abc`, along with the terminating byte; the label `SL5` points to the string:

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

  String labels have the form **SL***n*, where *n* is a number assigned by the compiler, beginning with 0 and increasing by 1 for each defined string. All strings used in a source module are defined at the end of the compiled assembly language module.

  The label **SL***n* represents the address of the string constant (a pointer to the string). Like all addresses of static objects, this address must be stored in the constant table in order to be accessed. Thus, in addition to storing the string itself in the .text section, the compiler uses the following directive statement to store the string's address in the constant table:

  ```
         .word  SLn
  ```

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

Note:

Each source module can have a maximum of 400 unique string constants; the code generator aborts with an error message if this limit is exceeded.

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

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

4.2.5 The Constant Table

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

- Integer constants that are wider than 16 bits.
- Floating-point constants that have exponents larger than 4 bits or mantissas larger than 11.
- Addresses of global variables.
- Addresses of string constants.

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

```
CONST:     .word     011223344h     ;32 bit constant
           .float    3.1459265      ;floating-point constant
           .word     _globvar       ;address of global
           .word     SL23           ;address of string
```

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

```
          LDI    @CONST+offset,R0
```

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

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

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

As with all autoinitialization, you can avoid the extra use of memory required for the .cinit section by using the –cr linker option and using a smart loader to perform the initialization directly from the object file. For more information about autoinitialization, refer to Section 4.10 on page 4-30.

Note:

1) The total size of the constant table in one module is limited to 1000 entries. If this limit is exceeded, the code generator aborts with an error message.

2) Note that the small memory model restricts the total size of the global data page, *including the constant tables*, to 64K words.

4.3 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 C compiler uses the following registers:

Table 4–1. List of the Registers the Compiler Uses

Register	Description
R0	Integer and floating-point expression register, also, scalar return values
R1	Integer and floating-point expression register
R2	Integer and floating-point expression register
R3	Integer and floating-point expression register
R4	Integer register variable
R5	Integer register variable
R6	Floating-point register variable
R7	Floating-point register variable
AR0	Pointer expression register
AR1	Pointer expression register
AR2	Pointer expression register
AR3	Frame pointer (FP)
AR4	Pointer register variable
AR5	Pointer register variable
AR6	Pointer register variable
AR7	Pointer register variable
IR0	Used for extended addressing on local frame
IR1	Used for extended addressing on local frame
SP	Stack pointer

4.3.1 Expression Analysis Registers

The compiler uses registers R0—R3 and AR0—AR2 to evaluate expressions and store 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

TMS320C30's efficient register addressing modes and to avoid unnecessary accesses of variables and constants.

When a function is called, the compiler forgets the contents of the expression registers. The contents of any register that is being used for temporary storage is saved off to the local frame before the function is called. This prevents the called function from ever having to save and restore expression registers.

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

4.3.2 Return Values

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

4.3.3 Register Variables

Specific registers are reserved for variables that are declared with the *register* storage class specifier. The *register* designation tells the compiler to store the associated variable in a register if possible, for efficient access. Register storage can be specified for any type of automatic variables, both function arguments and local variables. There are several registers for each type of register variable:

Register	Description
R4, R5	are used for *integer* register variables.
R6, R7	are used for *floating-point* register variables.
AR4—AR7	are used for *pointer* register variables.

These registers are allocated in the order that they are declared; for example, the first integer variable declared as register is assigned to R4, and the second is assigned to R5. If a function declares more register variables than the number of registers that are available for that type, the excess variables are treated as automatic variables.

Using register variables can significantly increase the efficiency of a function, especially when values are frequently assigned to a particular variable (`var = ...`).

Any function that uses register variables **must** save the contents of each register used on entrance to the function and restore them on exit. This ensures that a called function does not disrupt the register variables of the calling function.

Unused register variables can be freely manipulated using inline assembly language.

4.3.4 Other Registers

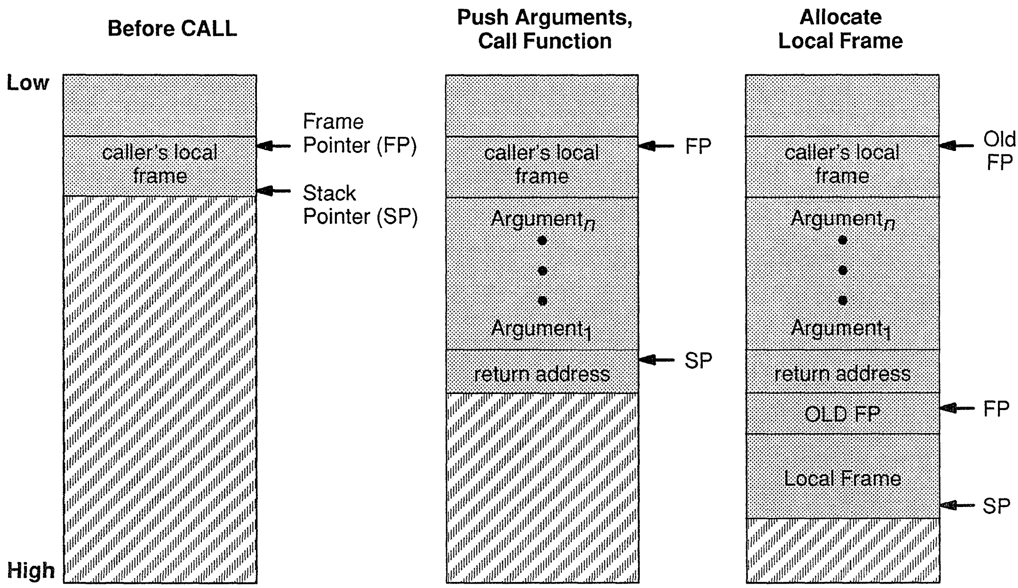
- The stack pointer (SP) and frame pointer (AR3) are used to manage the local frame.
- The page pointer (DP) is used to access global and static variables. Called functions must preserve the values in these registers.
- Index registers IR0 and IR1 are used for indirect addressing when an offset of more than 8 bits (±255) is required. They are treated like expression registers and need not be saved by called functions.
- The block-repeat registers (RS, RE, and RC) are used to copy structures. They need not be saved by called functions.

4.4 Function Structure and Calling Conventions

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

Figure 4–1 illustrates a typical function call. In this example, parameters are passed to the function and the function uses local variables.

Figure 4–1. Stack Use During a Function Call



4.4.1 Responsibilities of a Calling Function

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

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

2) The caller calls the function.

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

```
SUBI    n,SP
```

n is the number of argument words that were pushed.

4.4.2 Responsibilities of a Called Function

A called function must perform the following tasks.

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

Save as integers			Save as floating-point
R4	R5	AR4	R6
AR5	AR6	AR7	R7
FP			

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

2) It executes the code for the function.
3) It restores all saved registers.
4) If the function returns an integer, pointer, or float, it places the return value in R0. If the function returns a structure, refer to Section 4.4.5 on page 4-17.

4.4.3 Setting Up the Local Frame

Called C functions perform additional actions in order to manage the local frame. Note that if the function has no local variables, and no need for local temporary storage, these actions are not taken.

1) The called function sets up the local frame; this is the first action taken by the called function. The local frame is allocated as follows:
   a) The old frame pointer is saved on the stack.
   b) The new frame pointer is set to the current SP.
   c) The frame is allocated by adding its size to the SP.
2) Before returning, the called function deallocates the frame by subtracting its size from SP and restores the old FP by popping it.

4.4.4 Accessing Arguments and Local Variables

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

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

Note:

It is best to avoid using locals and arguments with offsets larger than 255 words. The sequence used to access such variables is:

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

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

4.4.5 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 in register AR0. To return a structure, the called function then copies the structure to the memory block that AR0 points to.

In this way, the caller can be "smart" about telling the called function where to return the structure. For example, in the statement `s = f()`, where `s` is a structure and `f` is a function that returns a structure, the caller can simply place the address of `s` in AR0 and call `f`. Function `f` then copies the return structure directly into `s`, performing the assignment automatically.

If the caller does not use the return value, AR0 is set to 0. This directs the called function not to copy the return structure.

You must be careful to properly declare functions that return structures both at the point where they are called (so the caller properly sets up AR0) and where they are defined (so the function knows to copy the result).

4.5 Interfacing C with Assembly Language

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

- Use separate modules of assembled code and link them with compiled C modules (see Section 4.5.1). This is the most versatile method.
- Use inline assembly language that is imbedded directly in the C source (see Section 4.5.2, page 4-22).
- Modify the assembly language code that the compiler produces (see Section 4.5.3, page 4-22).

4.5.1 Assembly Language Modules

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

1) All functions, whether they are written in C or assembly language, must follow the conventions outlined in Section 4.4, page 4-15).

2) You must preserve any dedicated registers that are modified by a function; dedicated registers include:

Dedicated Registers			
R4	R5	R6	R7
AR4	AR5	AR6	AR7
SP	FP (AR3)		

   All registers are saved as integers except R6 and R7, which are saved as floating-point values. Note that if the SP is used normally, it does not need to be explicitly preserved. In other words, the assembly function is free to use the stack as long as anything that is pushed is popped back off before the function returns (thus preserving SP).

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

3) Interrupt routines must save **all** the registers they use. Expression registers R0—R3 must be saved as complete 40-bit values, because they may contain either integers or floating-point values when the interrupt occurs. For more information about interrupt handling, refer to Section 4.6 on page 4-23.

4) When calling a C function from assembly language, push any arguments on the stack in reverse order. Pop them off after calling the function. When calling C functions, remember that only the dedicated registers listed above are preserved. C functions can change the contents of any other register.

5) Functions must return values correctly according to their C declarations. Integers, pointers, and floating-point values are returned in register R0, and structures are returned as described in Section 4.4.5 on page 4-17.

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

7) The compiler appends an underscore (_) to the beginning of all identifiers. In assembly language modules, you must use a prefix of _ for all objects that are to be accessible from C. For example, a C object named `x` is called `_x` in assembly. 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.

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

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

4.5.1.1 An Example of an Assembly Language Function

The example in Section 4.2 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.

Figure 4–2. An Assembly Language Function

(a) C program

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

main()
{
   int i;

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

(b) Assembly language program

```
FP  .set       AR3          ; FP is AR3
    .global    _asmfunc     ; Declare external function
    .global    _gvar        ; Declare external variable

_asmfunc:
    PUSH       FP           ; Save old FP
    LDI        SP,FP        ; Point to top of stack
    LDI        *-FP(2),R0   ; Load argument into R0
    LDP        _gvar        ; Set DP to page of gvar
                            ; (BIG MODEL ONLY)
    ADDI       @_gvar,R0    ; Add gvar to argument in R0
    POP        FP           ; Restore FP
    RETS
```

In the C program in Figure 4–2, the `extern` declaration of `asmfunc` is optional because the function returns an int. Like C functions, assembly functions need be declared only if they return non-integers.

In the assembly language code in Figure 4–2, note the underscores on all the C symbol names. Note also that the DP needs to be set only when accessing global variables in the big model. For the small model, the LDP instruction that loads the page pointer can be omitted.

4.5.1.2 Defining 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:

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

Figure 4–3 shows an example for accessing a variable defined in .bss.

Figure 4–3. 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 int var;         /* External variable */
    var = 1;                /* Use the variable  */
```

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

The first step is to define the object. It is helpful, but not necessary, to put it in its own initialized section. Declare a global label that points to the beginning of the object.

The object can be linked anywhere into the memory space. To access it in C, you must declare an additional C variable to point to the object. Initialize the pointer with the assembly language label declared for the object; remember to remove the underscore.

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

Figure 4–4. Accessing a Variable that is not Defined in .bss from C

(a) Assembly Language Program

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

(b) C Program

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

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

4.5.2 Inline Assembly Language

Within a C program, you can use the **asm statement** to inject a single line of assembly language into the assembly language file that the compiler creates. A series of asm statements places sequential lines of assembly language into the compiler output with no intervening code. For more information about the asm statement, refer to Section 3.8 on page 3-11.

Note:

Inserting jumps or labels into C code may produce unpredictable results by confusing the register-tracking algorithms that the code generator uses. The asm statement is provided so that you can access features of the hardware which would be otherwise inaccessible from C.

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

In addition, 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 an asterisk (*) as shown below:

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

4.5.3 Modifying Compiler Output

You can inspect and change the assembly language output that the compiler produces by compiling the source and then editing the output file before assembling it. The note in Section 4.5.2 about disrupting the C environment also apply to modification of compiler output.

4.6 Interrupt Handling

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

4.6.1 Saving Registers During Interrupts

When C code is interrupted, the interrupt routine must preserve the contents of **all** machine registers. A problem arises with the extended-precision registers used for expression analysis (R0—R3): these registers can contain either integer or floating-point values, and an interrupt routine cannot determine the type of value in a register. Thus, an interrupt routine must preserve **all 40 bits** of any of these registers that it modifies. This involves saving both the integer part (lower 32 bits) and the floating-point part (upper 32 bits). You can avoid this problem by not using these registers for handling interrupts.

The following code saves and restores all 40 bits of a register:

```
PUSH      R0     ; Save bottom 32 bits
PUSHF     R0     ; Save top 32 bits
   .
   .
   .
POPF      R0     ; Restore top 32 bits
POP       R0     ; Restore bottom 32 bits
```

If the interrupt routine modifies R6 or R7, which are reserved for the floating-point register variables, only the floating-point contents must be preserved. These registers can contain only floating-point values.

Any other registers that are modified by the interrupt routine can contain integers (or pointers) only, so only the integer part (lower 32 bits) must be preserved.

4.6.2 Using C Interrupt Routines

Interrupts can be handled directly with C functions by using a special naming convention. C interrupt functions have names with the following format:

`c_int`*nn*

nn is a two-digit interrupt number between 00 and 99 (for example, a valid interrupt routine name is `c_int01`). By following this convention for naming interrupt routines, you assure that the compiler uses the special register preservation requirements that are discussed in Section 4.6.1.

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** the expression registers if any other functions are called. This uses many extra instructions; if you are **sure** that a particular register will not be modified, you can hand-modify the compiled code so that an unused register is not saved and restored.

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

4.6.3 Assembly Language Interrupt Routines

Interrupts can also be handled with assembly language code, as long as the register conventions are followed. Like all assembly functions, interrupt routines can use the stack, access global C variables, and call C functions normally. When calling C functions, be sure that **all** nondedicated registers are preserved because the C function can modify any of them. Of course, dedicated registers need not be saved because they are preserved by the C function.

4.7 Expression Analysis

All C expressions are calculated using the registers designated for expression analysis:

- Registers **R0—R3** are used for expression evaluation.
- Registers **AR0—AR2** are used for indirection with pointers.

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

The compiler attempts to avoid using the address registers in evaluation because pipeline delays can result from using auxiliary registers for both computation and indirection. This is apparent in the code generated for pointer arithmetic, where the arithmetic is evaluated in R0—R3, then moved to an auxiliary register when the resulting pointer is actually used.

Floating-point expressions are evaluated using the on-chip floating-point hardware. In general, this means that all floating-point operations are carried out with full extended precision (40 bits). However, in some cases an extended-precision temporary result in a register must be saved off to memory, in which case only 32 bits of precision are preserved.

4.8 Runtime-Support Math Routines

The TMS320C30 MPYI (multiply integer) instruction does not perform full 32-bit multiplication; it uses only the lower 24 bits of each operand. Standard C requires full 32-bit multiplication. Therefore, a runtime-support function called MPY_I is provided to implement 32-bit integer multiplication. This function does not follow the standard C calling sequence; instead, operands are passed in registers R0 and R1. The 32-bit product is returned in R0. The compiler uses the TMS320C30 MPYI instruction only in cases where address arithmetic is performed (such as during array indexing); because no address can have more than 24 bits, a 24×24 multiply is sufficient. You can use the –mm option to force the compiler to use MPYI instructions for all integer multiplies.

Because the TMS320C30 has no division instructions, integer and floating-point division are performed via calls to additional runtime-support functions called DIV_I and DIV_F. Another function called MOD_I performs the integer modulo operation. Corresponding functions called DIV_U and MOD_U are used for unsigned integer division and modulo. Like MPY_I, these functions take their arguments from R0 and R1 and return the result in R0.

The runtime-support math functions can use volatile registers R0—R3 and the index registers IR0 and IR1 without saving them. Any other registers that are used must be saved. The versions of the functions supplied with the compiler use no additional registers.

The runtime-support math functions are written in assembly language. Object code for them is provided in the object library `rts.lib`. Any of these functions that your program needs are linked in automatically if you name `rts.lib` as input at link time.

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

Figure 4–5 summarizes the runtime-support math functions and names the files that contain the functions.

Figure 4–5. Summary of Runtime-Support Math Functions

Function	Description	Source File
DIV_F	Floating-point divide	`divf.asm`
DIV_I	Integer divide	`divi.asm`
DIV_U	Unsigned integer divide	`divu.asm`
MOD_I	Integer modulo	`modi.asm`
MOD_U	Unsigned integer modulo	`modu.asm`
MPY_I	32x32 Integer multiply	`mpyi.asm`

4.9 Optimization

The TMS320C30 C compiler was designed with two major goals in mind:

- For general purpose C code, the TMS320C30 C compiler produces compiled code that performs nearly as well as hand-coded assembly language.
- For critical DSP algorithms, the TMS320C30 C compiler provides a simple and accessible programming environment so that applications demanding high performance can be implemented in assembly language.

The compiler performs a wide variety of optimizations to improve the efficiency of compiled code. The degree of optimization relative to hand-coded assembly language for a given program is *extremely* dependent on how the program is written; if the code is written specifically with the C30 compiler in mind, the generated code can be nearly as efficient as assembly language.

The following list describes some of the optimizations and highlights particular strengths of the compiler:

- **Register Variables**

  By using register variables, the compiler generates *excellent* code for expressions involving these variables. Register variables are particularly valuable as pointers.

- **Register Tracking**

  The compiler tracks the contents of registers so it avoids reloading values if they are used again soon. Variables, constants, and structure references *(a.b)* are tracked through straight-line code and forward branches.

- **3-Operand Instructions**

  By using 3-operand instructions whenever possible, the compiler preserves the contents of the registers and allows more flexibility in addressing. These instructions are particularly effective in conjunction with register variables.

- **Algebraic Reordering**

  The compiler reorders expressions into algebraic equivalents to allow optimal evaluation. For example: *–(a + b)*, which takes 3 instructions to evaluate, is written as *–a –b*, which only takes 2 instructions.

- **Jump Optimizations**

  The compiler unwinds jumps to jumps and eliminates dead code (unlabeled code following an unconditional jump).

- **Loop Rotation**

  The compiler evaluates loop conditionals at the top *and* bottom of loops, saving a costly extra jump into or out of a loop. In cases of simple counting loops *(for (i= 0; i< 10;. . .))* the initial entry conditional check is optimized out.

- **Delayed Branches**

  Where possible, the compiler uses delayed branches for unconditional branches, avoiding pipeline delays caused by standard branches.

- **Parallel Instructions**

  Because of the restrictive addressing requirements of the parallel instructions, it is difficult for the compiler to take advantage of them. However, in cases where two adjacent instructions fit the addressing requirements, they are combined in parallel instructions. Also, the compiler uses parallel instructions for structure move operations.

- **Autoincrement Addressing**

  For pointer expressions of the form **p++, *p––, *++p,* or **––p,* the compiler uses efficient autoincrement addressing modes.

Note:

If you use the **–g** option to generate symbolic debugging information, many of these optimizations are disabled because they disrupt the debugger. If you want to use symbolic debugging and still generate fully optimized code, use the **–mn** option on CL30; –mn re-enables the optimizations disabled by –g.

4.10 System Initialization

Before you can run a C program, the C runtime environment must be created. This task is performed by the C boot routine, which is a function called c_int00. The runtime-support source library contains the source to this routine in a module named `boot.asm`.

The c_int00 function can be called by reset hardware to begin running the system. The function is in the runtime support library (`rts.lib`) and must be combined with the C object modules at link time. This occurs by default when you use the –c or –cr option in the linker and include `rts.lib` as one of the linker input files. When C programs are linked, the linker sets the entry point value in the executable output module to the symbol `c_int00`.

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

1) Defines a section called .stack for the system stack and sets up the initial stack and frame pointers.

2) Autoinitializes global variables by copying the data from the initialization tables in .cinit to the storage allocated for the variables in .bss. In the small model, the constant tables are also copied from .cinit to .bss.

   In the RAM initialization model, a loader performs this step before the program runs (it is not performed by the boot routine).

3) *Small memory model only*—sets up the page pointer DP to point to the global storage page in .bss.

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 in order to correctly initialize the C environment.

4.10.1 Autoinitialization of Variables and Constants

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

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

Note:

In standard C, global and static variables that are not explicitly initialized are set to 0 before program execution. The TMS320C30 C compiler does not adhere to this convention. Any variable which must have an initial value of 0 must be explicitly initialized.

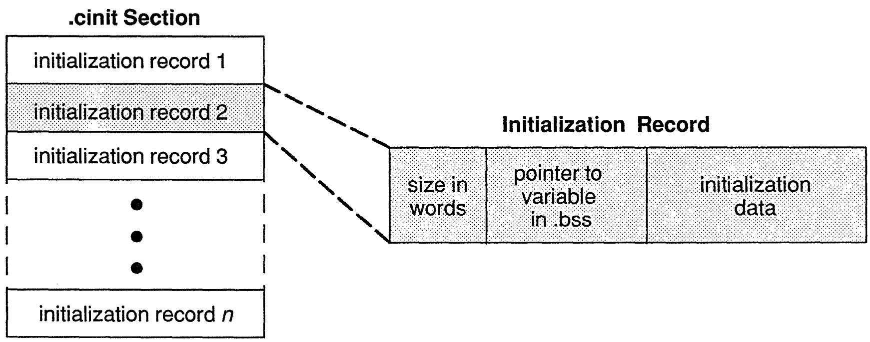
In the small memory model, any tables of long constant values or constant addresses must also be copied into the global data page at this time. Data for these tables is incorporated into the initialization tables in .cinit and thus is automatically copied at initialization time.

There are two methods for copying the autoinitialization data into memory; these methods are called the RAM and ROM models of autoinitialization. Section 4.10.1.1 describes the format of the initialization tables, Section 4.10.1.2 describes the RAM model of initialization, and Section 4.10.1.3 describes the ROM model of initialization.

4.10.1.1 Initialization Tables

The tables in .cinit consist of variable size initialization records. Figure 4–6 shows the format of the .cinit section and the initialization records.

Figure 4–6. Format of Initialization Records in the .cinit Section



- The first field of an initialization record is the size (in words) of the initialization data.
- The second field is the starting address of the area within the .bss section, where the initialization data must be copied. (This field points to a variable's space in .bss.)
- These first two fields are followed by one or more words of data. During autoinitialization, this data is copied to the specified address in .bss.

Each variable that must be autoinitialized has an initialization record in the .cinit section.

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 word)
   .word     _i            ; Address in .bss
   .word     23            ; Data

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

The .cinit must contain **only** initialization tables in this form. 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.

4.10.1.2 RAM Initialization Model

The RAM model, specified with the –cr linker option, allows variables to be initialized at *load time* instead of at run time. This enhances system performance by reducing boot time and by saving the memory that would ordinarily be used by the initialization tables.

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` section 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 run-time initialization is performed at boot time.

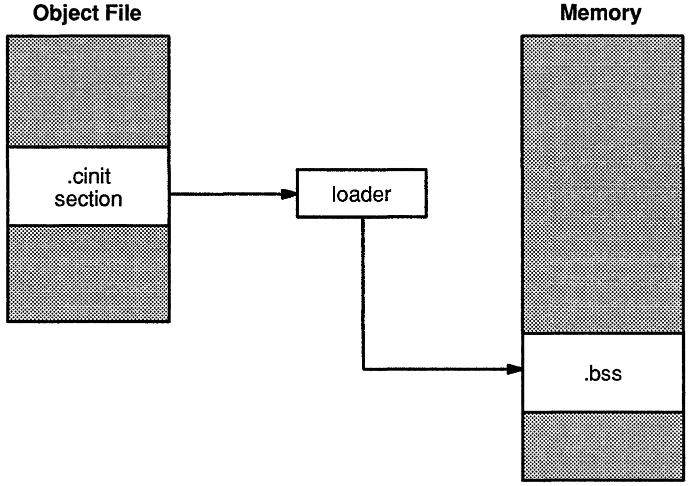
Note that you must use a smart loader to take advantage of the RAM model. When the program is loaded, the loader must be able to:

- 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 4–7 illustrates the RAM model of autoinitialization.

Figure 4–7. RAM Model of Autoinitialization



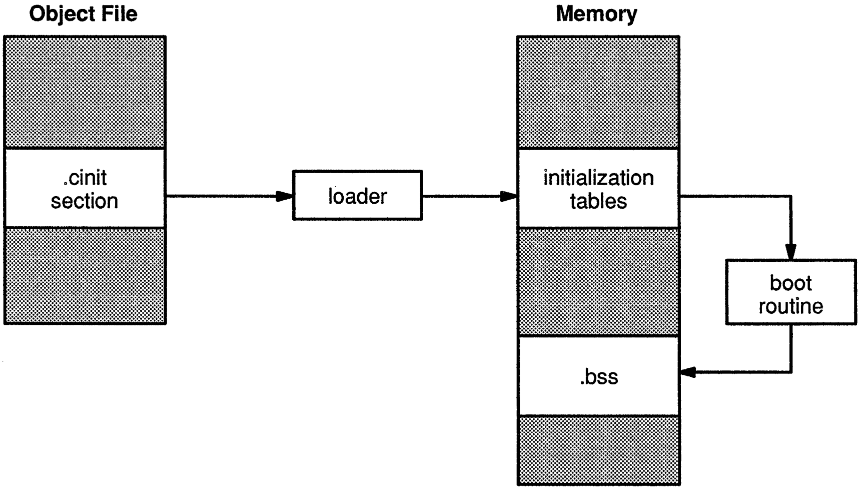
4.10.1.3 ROM Initialization Model

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

In this method, global variables are initialized at *run time*. 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 .bss. This allows initialization data to be stored in ROM and then copied to RAM each time the program is started.

Figure 4–8 illustrates the ROM model of autoinitialization.

Figure 4–8. ROM Model of Autoinitialization



Chapter 5

Runtime-Support Functions

Some of the tasks that a C program must perform (such as floating-point arithmetic, string operations, and dynamic memory allocation) are not part of the C language. The runtime-support functions, which are included with the C compiler, are standard functions that perform these tasks. The runtime-support library `rts.lib` contains the object code for each of the functions described in this chapter; the library `rts.src` contains the source to these functions as well as to other functions and routines. If you use any of the runtime-support functions, be sure to include `rts.lib` as linker input when you link your C program.

This chapter is divided into three parts:

- Part 1 describes header files and discusses their functions.
- Part 2 summarizes the runtime-support functions according to category.
- Part 3 is an alphabetical reference.

Topics in this chapter include:

5.1 Header Files

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

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

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

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

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

```
#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.

Sections 5.1.1 through 5.1.10 describe the header files that are included with the TMS320C30 C compiler. Section 5.2, page 5-9, lists the functions that these headers declare.

5.1.1 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, 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 the assert macro is turned off and does nothing. If NDEBUG *is not* defined, then the assert macro is enabled.

The assert macro is defined as follows:

```
#ifdef NDEBUG
#define assert(ignore)
#else
#define assert(expr) \
if (!(expr)) {printf("Assertion failed,(expr), file s,\
   line d\n", __FILE__, __LINE__); abort(); }
#endif
```

5.1.2 Character Typing and Conversion (ctype.h)

The ctype.h header declares functions that test (type) and convert characters.

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

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

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

The ctype.h header also contains macro definitions that perform these same operations; the macros run faster than the corresponding functions. The typing macros expand to a lookup operation in an array of flags (this array is defined in ctype.c). The macros have the same name as the corresponding functions, but each macro is prefixed with an underscore (for example, *_isdigit*).

5.1.3 Limits (float.h and limits.h)

The float.h and limits.h headers define macros that expand to useful limits and parameters of the TMS320C30's numeric representations. Table 5–1 and Table 5–2 list these macros and the limits they are associated with.

Table 5–1. Macros that Supply Integer Type Range Limits **(`limits.h`)**

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

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

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

Key to prefixes:
FLT_ applies to type float
DBL_ applies to type double
LDBL_ applies to type long double

5.1.4 Floating-Point Math (`math.h`)

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

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

5.1.5 Error Reporting (errno.h)

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

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

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

5.1.6 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:

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

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

5.1.7 Standard Definitions (stddef.h)

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

- *ptrdiff_t,* a signed integer type that is the data type resulting from the subtraction of two pointers; **and**
- *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), **and**
- The *offsetof(type, identifier)* macro, which expands to an integer that has type size_t. The result is the value of an offset in bytes to a structure member (*identifier*) from the beginning of its structure (*type*).

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

5.1.8 General Utilities (`stdlib.h`)

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

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

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

- Memory management functions that allow you to allocate and deallocate packets of memory. The amount of memory that these functions can use is defined by the constant `__SYSMEM_SIZE` in the runtime-support module `sysmem.asm`. (This module is archived in the file `rts.src`.) By default, the amount of memory available for allocation is 2048 words. You can change this amount by modifying `__SYSMEM_SIZE` and reassembling `sysmem.asm`.
- 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.

5.1.9 String Functions (`string.h`)

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

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

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

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

5.1.10 Time Functions (time.h)

The `time.h` header declares one macro, several types, and functions that manipulate dates and time. The functions deal with several types of time:

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

The `time.h` header declares one macro, *CLK_TCK*, which is the number per second of the value returned by the clock function.

The header declares three types:

- *clock_t*, an arithmetic type that represents time;
- *time_t*, an arithmetic type that represents time, **and**
- *tm* is a structure that holds the components of calendar time, 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 –      */
```

`tm_isdst` can have one of three values:

- A *positive* value if Daylight Savings Time is in effect.
- *Zero* if Daylight Savings Time is not in effect.
- A *negative* value if the information is not available.

Note:

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

5.2 Summary of Runtime-Support Functions and Macros

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

Error Message Macro
(Header File: assert.h)

Macro and Syntax	*Description*
`void` **`assert`** `(expression)` `int expression;`	Inserts diagnostic messages into programs

Character Typing Conversion Functions
(Header File: ctype.h)

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

Floating-Point Math Functions
(Header File: `math.h`)

Function and Syntax	*Description*
`double` **`acos`** `(x)` `double x;`	Returns the arc cosine of `x`
`double` **`asin`** `(x)` `double x;`	Returns the arc sine of `x`
`double` **`atan`** `(x)` `double x;`	Returns the arc tangent of `x`
`double` **`atan2`** `(y,x)` `double y,x;`	Returns the inverse tangent of `y/x`
`double` **`ceil`** `(x)` `double x;`	Returns the smallest integer greater than or equal to `x`
`double` **`cos`** `(x)` `double x;`	Returns the cosine of `x`
`double` **`cosh`** `(x)` `double x;`	Returns the hyperbolic cosine of `x`
`double` **`exp`** `(x)` `double x;`	Returns the exponential function of `x`
`double` **`fabs`** `(x)` `double x;`	Returns the absolute value of `x`
`double` **`floor`** `(x)` `double x;`	Returns the largest integer less than or equal to `x`
`double` **`fmod`** `(x, y)` `double x, y;`	Returns the floating-point remainder of `x/y`

`double` **`frexp`** `(value,exp)`
`double value;`
`int *exp;`
	Breaks `value` into a normalized fraction and an integer power of 2

`double` **`ldexp`** `(x, exp)`
`double x;`
`int exp;`
	Multiplies `x` by an integer power of 2

`double` **`log`** `(x)` `double x;`	Returns the natural logarithm of `x`
`double` **`log10`** `(x)` `double x;`	Returns the base-10 logarithm of `x`

`double` **`modf`** `(value, iptr)`
`double value;`
`int *iptr;`
	Breaks `value` into into a signed integer and a signed fraction

`double` **`pow`** `(x, y)` `double x, y;`	Returns `x` raised to the power `y`
`double` **`sin`** `(x)` `double x;`	Returns the sine of `x`
`double` **`sinh`** `(x)` `double x;`	Returns the hyperbolic sine of `x`

Floating-Point Math Functions
(continued)

Function and Syntax	*Description*
`double sqrt (x)` `double x;`	Returns the nonnegative square root of `x`
`double tan (x)` `double x;`	Returns the tangent of `x`
`double tanh (x)` `double x;`	Returns the hyperbolic tangent of `x`

Variable Argument Functions and Macros
(Header File: `stdarg.h`)

Function/Macro and Syntax	*Description*
`type va_arg (ap, type)` `va_list ap;`	Accesses the next argument of type `type` in a variable-argument list
`void va_end (ap)` `va_list ap;`	Resets the calling mechanism after using `va_arg`
`void va_start(ap, parmN)` `va_list ap;`	Initializes ap to point to the first operand in the variable-argument list

General Utilities
(Header File: `stdlib.h`)

Function and Syntax	*Description*
`int abs (j)` `int j;`	Returns the absolute value of `j`
`void abort ()`	Terminates a program abnormally
`void atexit (fun)` `void (*fun)();`	Registers the function pointed to by `fun`, to be called with out arguments at normal program termination
`double atof (nptr)` `char *nptr;`	Converts a string to a floating-point value
`int atoi (nptr)` `char *nptr;`	Converts astring to an integer value
`long int atol (nptr)` `char *nptr;`	Converts astring to a long integer

`void *bsearch (key, base, nmemb, size, compar)`
`void *key, *base;`
`size_t nmemb, size;`
`int (*compar)();`
	Searches through an array of `nmemb` objects for the object that `key` points to

`void *calloc (nmemb, size)` `size_t nmemb, size`	Allocates and clears memory for `nmemb` objects, each of `size` bytes

General Utilities
(continued)

Function and Syntax	*Description*
`div_t` **`div`** `(numer, denom)` `int numer, denom`	Divides `numer` by `denom`
`void` **`exit`** `(status)` `int status;`	Terminates a program normally
`void` **`free`** `(ptr)` `void *ptr;`	Deallocates memory space allocated by `malloc`, `calloc`, or `realloc`
`long int` **`labs`** `(j)` `long int j;`	Returns the absolute value of `j`
`ldiv_t` **`ldiv`** `(numer, denom)` `long int numer, denom`	Divides `numer` by `denom`

`int` **`ltoa`** `(n, buffer)`
`long n;`
`char *buffer;`
	Converts `n` to the equivalent string

`void` **`*malloc`** `(size)` `size_t size`	Allocates memory for an object of `size` bytes
`void` **`minit`** `()`	Resets all the memory previously allocated by `malloc`, `calloc`, or `realloc`

`char` **`*movmem`** `(src,dest,count)`
`char *src, *dest;`
`int count;`
	Moves `count` bytes from `src` to `dest`

`void` **`qsort`** `(base, nmemb, size, compar)`
`void *base;`
`size_t nmemb, size;`
`int (*compar)();`
	Sorts an array of `nmemb` members; `base` points to the first member of the unsorted array, and `size` specifies the size of each member

`int` **`rand`** `()`	Returns a sequence of pseudo-random integers in the range 0 to RAND_MAX

`void` **`*realloc`** `(ptr, size)`
`void *ptr;`
`size_t size;`
	Changes the size of an allocated memory space

`void` **`srand`** `(seed)` `unsigned int seed;`	Resets the random number generator
`double` **`strtod`** `(nptr, endptr)` `char *nptr, **endptr;`	Converts a string to a floating-point value

`long int` **`strtol`** `(nptr, endptr, base)`
`char *nptr,**endptr;`
`int base;`
	Converts a string to a long integer

`unsigned long int` **`strtoul`**
`char *nptr, **endptr;`
`int base;`
	Converts a string to an unsigned long integer

String Functions
(Header File: string.h)

Function and Syntax	*Description*

`void *memchr(s, c, n)`
`void *s;`
`int c;`
`size_t n;`
	Finds the first occurrence of c in the first n characters of s

`int memcmp (s1, s2, n)`
`void *s1, *s2;`
`size_t n;`
	Compares the first n characters of s1 to s2

`void *memcpy (s1, s2, n)`
`void *s1, *s2;`
`size_t n;`
	Copies n characters from s1 to s2

`void *memmove (s1, s2, n)`
`void *s1, *s2;`
`size_t n;`
	Moves n characters from s1 to s2

`void *memset (s, c, n)`
`void *s;`
`int c;`
`size_t n;`
	Copies the value of c into the first n characters of s

`char *strcat (s1, s2)` `char *s1, *s2;`	Appends s1 to the end of s2

`char *strchr (s, c)`
`char *s;`
`int c;`
	Finds the first occurrence of character c in s

`int strcmp (s1, s2)`
`char *s1, *s2;`
`is greater than 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 (s1, s2)` `char *s1, *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 (s1, s2)` `char *s1, *s2;`	Copies string s2 into s1
`size_t strcspn (s1, s2)` `char *s1, *s1;`	Returns the length of the initial segment of s1 that is made up entirely of characters that are not in s2
`char *strerror (errnum)` `int errnum;`	Maps the error number in errnum to an error message string
`size_t strlen (s)` `char *s;`	Returns the length of a string

`char *strncat (s1, s2, n)`
`char *s1, *s2;`
`size_t n;`
	Appends up to n characters from s1 to s2

`int *strncmp (s1, s2, n)`
`char *s1, *s2;`
`size_t n;`
	Compares up to n characters in two strings

String Functions
(continued)

Function and Syntax	*Description*

`char *strncpy (s1, s2, n)`
`char *s1, *s2;`
`size_t n;`
	Copies up to `n` characters of a `s2` to `s1`

`char *strpbrk (s1, s2)` `char *s1, *s2;`	Locates the first occurrence in `s1` of *any* character from `s2`

`char *strrchr (s ,c)`
`char *s;`
`int c;`
	Finds the last occurrence of character `c` in `s`

`size_t strspn (s1, s2)` `char *s1, *s2;`	Returns the length of the initial segment of `s1`, which is entirely made up of characters from `s2`
`char *strstr (s1, s2)` `char *s1, *s2;`	Finds the first occurrence of a `s2` to `s1`
`char *strtok (s1, s2)` `char *s1, *s2;`	Breaks `s1` into a series of tokens, each delimited by a character from `s2`

Time Functions
(Header File: `time.h`)

Function and Syntax	*Description*
`char *asctime (timeptr)` `struct tm *timeptr;`	Converts a time to a string
`clock_t clock ()`	Determines the processor time used
`char *ctime (timeptr)` `struct tm *timeptr;`	Converts calendar time to local time
`double difftime (time1,time0)` `time_t time1, tim0;`	Returns the difference between two calendar times
`struct tm *gmtime (timer)` `time_t *timer;`	Converts calendar time to Greenwich Mean Time
`struct tm *localtime (timer)` `time_t *timer;`	Converts calendar time to local time
`time_t mktime (timeptr)` `struct tm *timeptr;`	Converts local time to calendar time

`size_t strftime (s, maxsize, format, timeptr)`
`char *s, *format;`
`size_t maxsize;`
`struct tm *timeptr;`
	Formats a time into a character string

`time_t time (timer)` `time_t *timer;`	Returns the current calendar time

5.3 Functions Reference

The remainder of this chapter is a reference. Generally, the functions are organized alphabetically, one function per page; however, related functions (such as isalpha and isascii) are presented together on one page. Here's an alphabetical table of contents for the functions reference:

Syntax

```
#include <stdlib.h>
```

`void abort( )`

Defined in `exit.c` in `rts.src`

Description The abort function usually terminates a program with an error code. The TMS320C30 implementation of the abort function calls the exit function with a value of 0, and is defined as follows:

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

This makes the abort function functionally equivalent to the exit function.

Syntax

```
#include <stdlib.h>

int abs(j)
   int j;

long int labs(k)
   long int k;
```

Defined in `abs.c` in `rts.src`

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

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

Since *int* and *long int* are functionally equivalent types in TMS320C30 C, the abs and labs functions are also functionally equivalent.

Syntax

```
#include <math.h>

double acos(x)
   double x;
```

Defined in `asin.obj` in `rts.lib`

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

Syntax

```
#include <time.h>
char *asctime(timeptr)
   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 Section 5.1.10 on page 5-8.

Syntax

```
#include  <math.h>

double asin(x)
   double x;
```

Defined in asin.obj in rts.lib

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 */
```

Syntax

```
#include  <assert.h>
void  assert(expression)
   int   expression;
```

Defined in `assert.h` as macros

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

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

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

```
#define assert(ignore)
```

If NDEBUG is not defined when `assert.h` is included, then the assert macro is defined as:

```
#define assert(expr) \
if (!(expr)) {
      printf("Assertion failed, (expr), file %s,
            line %d\n", __FILE__ __LINE__)
   abort(); }
```

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;
```

Syntax

```
#include  <math.h>
double atan(x)
   double x;
```

Defined in atan.obj in rts.lib

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 */
```

Syntax

```
#include  <math.h>

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

Defined in `atan.obj` in rts.lib

Description The atan2 function returns the inverse tangent of `y/x`. The function uses the signs of the arguments to determine the quadrant of the return value. Both arguments cannot be 0. The return value is an angle in the range [$-\pi$,π] radians.

Example

```
double rvalu, rvalv;
double radians;

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

Syntax

```
#include <stdlib.h>

void atexit(fun)
   void (*fun)();
```

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.

Syntax

```
#include <stdlib.h>
double atof(nptr)
   char *nptr;
int atoi(nptr)
   char *nptr;
long int atol(nptr)
   char *nptr;
```

Defined in `atof.c` and `atoi.c` in `rts.src`

Description Three functions convert strings to numeric representations:

- The **atof** function converts a string to 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 to an integer. Argument `nptr` points to the string; the string must have the following format:

  [space] [sign] digits

- The **atol** function converts a string to 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 newline. Following the space is an optional *sign*, and then *digits* that represent the integer portion of the number. The fractional part of the number follows, then the exponent, including an optional *sign*.

The first character that cannot be part of the number terminates the string.

Since *int* and *long* are functionally equivalent in TMS320C30 C, the atoi and atol functions are also functionally equivalent.

The functions do not handle any overflow resulting from the conversion.

Syntax

```
#include <stdlib.h>

void *bsearch(key, base, nmemb, size, compar)
   void   *key, *base;
   size_t nmemb, size;
   int    (*compar)();
```

Defined in `bsearch.c` in `rts.src`

Description The bsearch function searches through an array of `nmemb` objects for a member that matches the object that `key` points to. Argument `base` points to the first member in the array; `size` specifies the size (in bytes) of each member.

The contents of the array must be in ascending, sorted order. If a match is found, the function returns a pointer to the matching member of the array; if no match is found, the function returns a null pointer (0).

Argument `compar` points to a function that compares `key` to the array elements. The comparison function should be declared as:

```
int  cmp(ptr1, ptr2)
   void  *ptr1, *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`.

Syntax

```
#include <stdlib.h>

void *calloc(nmemb, size)
   size_t nmemb;  /* number of items to allocate */
   size_t size;   /* size (in bytes) of each item */
```

Defined in memory.c in rts.src

Description The calloc function allocates size bytes for each of nmemb objects and returns a pointer to the space. The function initializes the allocated memory to all 0s. If it cannot allocate the memory (that is, if it runs out of memory), it returns a null pointer (0).

The memory that calloc uses is in a special memory pool or heap. An assembly language module called sysmem.asm defines this memory pool as uninitialized named section called .sysmem. The constant _ _SYSMEM_SIZE defines the size of the heap as 2048 words. If necessary, you can change the size of the heap by changing the value of _ _SYSMEM_SIZE and reassembling sysmem.asm. For more information, refer to Section 4.1.4, Dynamic Memory Allocation, on page 4-6.

Example This example uses the calloc routine to allocate and clear 10 bytes.

```
prt = calloc (10,2)  ; /*Allocate and clear 20 bytes */
```

Syntax

```
#include <math.h>

double ceil(x)
     double x;
```

Defined in `floor.obj` in `rts.lib`

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 */
```

Syntax

```
#include  <time.h>
clock_t  clock()
```

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:

The clock function is target-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 Section 5.1.10 on page 5-8.

Syntax

```
#include  <math.h>
double cos(x)
   double x;
```

Defined in sin.obj in rts.lib

Description The cos function returns the cosine of a floating-point number, x. x is an angle expressed in radians. An argument with a large magnitude may produce a result with little or no significance.

Example

```
double radians, cval;    /* cos returns cval */
radians = 3.1415927;
cval = cos(radians);     /* return value = -1.0 */
```

Syntax

```
#include  <math.h>
double cosh(x)
   double x;
```

Defined in `sinh.obj` in `rts.lib`

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 */
```

Syntax

```
#include  <time.h>

char  *ctime(timer)
   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 with that broken-down time as an argument.

For more information about the functions and types that the `time.h` header declares, refer to Section 5.1.10 on page 5-8.

Syntax

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

```
double  difftime(time1, time0)
   time_t  time1, 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 Section 5.1.10 on page 5-8.

Syntax

```
#include <stdlib.h>
div_t  div(numer, denom)
   int  numer, denom;
ldiv_t ldiv(numer, denom)
   long int  numer, denom;
```

Defined in `div.c` in `rts.src`

Description Two functions support integer division by returning `numer` divided by `denom`. You can use these functions to get both the quotient and the remainder in a single operation.

- The **div** function performs *integer* division. The input arguments are integers; the function returns the quotient and the remainder in a structure of type div_t. The structure is defined as follows:

```
typedef struct
{
   int  quot;        /*  quotient */
   int  rem;         /* remainder */
} div_t;
```

- The **ldiv** function performs *long integer* division. The input arguments are long integers; the function returns the quotient and the remainder in a structure of type ldiv_t. The structure is defined as follows:

```
typedef struct
{
   long int  quot;      /*  quotient */
   long int  rem;       /* remainder */
} ldiv_t;
```

If the division produces a remainder, then 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.

Since ints and longs are equivalent types in TMS320C30 C, these functions are also equivalent.

Syntax

```
#include <stdlib.h>

void exit(status)
   int status;
```

Defined in exit.c in rts.src

Description The exit function terminates a program normally. All functions registered by the atexit function are called, in reverse order of their registration.

You can modify the exit function to perform application-specific shutdown tasks. The unmodified function simply runs in an infinite loop until the system is reset.

Note that the exit function cannot return to its caller.

Syntax

```
#include <math.h>

double exp(x)
   double x;
```

Defined in exp.obj in rts.lib

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 */
```

Syntax

```
#include  <math.h>
double fabs(x)
   double x;
```

Defined in `fabs.obj` in `rts.lib`

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 */
```

Syntax

```
#include  <math.h>

double floor(x)
   double x;
```

Defined in `floor.obj` in `rts.lib`

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 */
```

Syntax

```
#include <math.h>

double fmod(x, y)
    double x, y;
```

Defined in `fmod.obj` in `rts.lib`

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 */
```

Syntax

```
#include  <stdlib.h>

void  free(ptr)
   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 Section 4.1.4, Dynamic Memory Allocation, on page 4-6.

Example This example allocates 10 bytes and then frees them.

```
char *x;
x = malloc(10);          /*  allocate 10 bytes   */
free(x);                 /*  free 10 bytes       */
```

Syntax

```
#include  <math.h>

double frexp(value, exp)
   double value;    /* input floating-point number */
   int    *exp;     /* pointer to exponent */
```

Defined in `frexp.obj` in `rts.lib`

Description The frexp function breaks a floating-point number into a normalized fraction and an integer power of 2. The function returns a value with a magnitude in the range (1/2,1) or 0, so that `value` = = `x` × 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 */
```

Syntax

```
#include  <time.h>

struct tm  *gmtime(timer)
   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 Section 5.1.10 on page 5-8.

Syntax

```
#include <ctype.h>

   int  isalnum(c)
      char c;
   int isalpha(c)
      char c;
   int isascii(c)
      char c;
   int iscntrl(c)
      char c;
   int isdigit(c)
      char c;
   int isgraph(c)
      char c;
   int islower(c)
      char c;
   int isprint(c)
      char c;
   int ispunct(c)
      char c;
   int isspace(c)
      char c;
   int isupper(c)
      char c;
   int isxdigit(c)
      char c;
```

Defined in `isxxx.c` and `ctype.c` in `rts.src`
Also defined in `ctype.h` as macros

Description These functions test a single argument `c` to see if it is a particular type of character—alphabetic, alphanumeric, numeric, ASCII, etc. If the test is true (the character is the type of character that it was tested to be), the function returns a nonzero value; if the test is false, the function returns 0. 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 between 0—127).

iscntrl identifies control characters (ASCII characters 0—31 and 127).

isdigit identifies numeric characters ('0'— '9')

isgraph identifies any non-space character.

islower identifies lowercase alphabetic ASCII characters.

isprint identifies printable ASCII characters, including spaces (ASCII characters 32—126).

ispunct identifies ASCII punctuation characters.

isspace identifies ASCII spacebar, tab (horizontal or vertical), carriage return, formfeed, and newline 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.

Syntax

```
#include <math.h>
double ldexp(x, exp)
   double x;
   int    exp;
```

Defined in `ldexp.obj` in `rts.lib`

Description The ldexp function multiplies a floating-point number by a power of 2 and returns $x \times 2^{exp}$. `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 */
```

Syntax

```
#include  <time.h>
struct tm  *localtime(timer)
   time_t  *timer;
```

Defined in `localtime.c` in `rts.src`

Description The local time 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 Section 5.1.10 on page 5-8.

Syntax

```
#include  <math.h>

double log(x)
   double x;
```

Defined in `log.obj` in `rts.lib`

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 */
```

Syntax

```
#include  <math.h>
double log10(x)
   double x;
```

Defined in `log.obj` in `rts.lib`

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 */
```

Syntax

```
#include  <stdlib.h>

int  ltoa(n, buffer)
   long   n;        /* number to convert          */
   char   *buffer;  /* buffer to put result in    */
```

Defined in `ltoa.c` in `rts.src`

Description The ltoa function converts a long integer to the equivalent ASCII string. 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`.

Syntax

```
#include  <stdlib.h>

void   *malloc(size)
   size_t size;     /* size of block in bytes */
```

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. An assembly language module called sysmem.asm defines this memory pool as uninitialized named section called .sysmem. The constant _ _SYSMEM_SIZE defines the size of the heap as 2048 words. If necessary, you can change the size of the heap by changing the value of _ _SYSMEM_SIZE and reassembling sysmem.asm. For more information, refer to Section 4.1.4, Dynamic Memory Allocation, on page 4-6.

Syntax

```
#include <string.h>

void    *memchr(s, c, n)
   void   *s;
   char   c;
   size_t n;
```

Defined in memchr.c in rts.src

Description The memchr function finds the first occurrence of c in the first n characters of the object that s points to. If the character is found, memchr returns a pointer to the located character; otherwise, it returns a null pointer (0).

The memchr function is similar to strchr, except the object that memchr searches can contain values of 0, and c can be 0.

Syntax

```
#include <string.h>

int memcmp(s1, s2, n)
   void   *s1, *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 the objects that memcmp compares can contain values of 0.

Syntax

```
#include  <string.h>
void  *memcpy(s1, s2, n)
   void   *s1, *s2;
   size_t n;
```

Defined in memmov.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 the objects that memcpy copies can contain values of 0.

Syntax

```
#include  <string.h>

void  *memmove(s1, s2, n)
   void   *s1, *s2;
   size_t n;
```

Defined in `memmov.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.*

Syntax

```
#include <string.h>
void  *memset(s, c, n)
   void   *s;
   char   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.

Syntax

```
#include <stdlib.h>
void minit()
```

Defined in memory.c in rts.src

Description The minit function resets all the space that was previously allocated by calls to the malloc, calloc, or realloc functions.

Note:

Calling the minit function makes **all** the memory space in the heap available again. **Any objects that you allocated previously will be lost; don't try to access them.**

The memory that minit uses is in a special memory pool or heap. An assembly language module called `sysmem.asm` defines this memory pool as uninitialized named section called `.sysmem`. The constant `__SYSMEM_SIZE` defines the size of the heap as 2048 words. If necessary, you can change the size of the heap by changing the value of `__SYSMEM_SIZE` and reassembling `sysmem.asm`. For more information, refer to Section 4.1.4, Dynamic Memory Allocation, on page 4-6.

Syntax

```
#include  <time.h>

time_t  *mktime(timeptr)
   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.

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);

printf ("result is %s\n", wday[time_str.tm_wday]);

/* After calling this function, time_str.tm_wday
    contains the day of the week for July 4, 2001 */
```

For more information about the functions and types that the `time.h` header declares, refer to Section 5.1.10 on page 5-8.

Syntax

```
#include  <math.h>

double modf(value, iptr)
   double value;
   int    *iptr;
```

Defined in `modf.obj` in `rts.lib`

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.              */
```

Syntax

```
#include  <stdlib.h>
char *movmem(src,dest,count)
   char *src ;      /* source address              */
   char *dest;      /* destination address         */
   char count;      /* number of bytes to move     */
```

Defined in `movmem.c` in `rts.src`

Description The movmem function moves `count` bytes of memory from the object that `src` points to into the object that `dest` points to. The source and destination areas can be overlapping.

Syntax

```
#include  <math.h>

double pow(x, y)
   double  x, y;  /* Raise x to power y */
```

Defined in `pow.obj` in `rts.lib`

Description The pow function returns `x` raised to the power `y`. A domain error occurs if `x =` 0 and `y` ≤ 0, or if `x` is negative and `y` is not an integer. A range error may occur.

Example

```
double x, y, z;

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

Syntax

```
#include  <stdlib.h>
void  qsort  (base, nmemb, size, compar)
   void   *base;
   size_t nmemb, size;
   int    (*compar)();
```

Defined in `qsort.c` in `rts.src`

Description The qsort function sorts an array of `nmemb` members. Argument `base` points to the first member of the unsorted array; argument `size` specifies the size of each member.

This function sorts the array in ascending order.

Argument `compar` points to a function that compares `key` to the array elements. The comparison function should be declared as:

```
int  cmp(ptr1, ptr2)
   void  *ptr1, *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`.

Syntax

```
#include <stdlib.h>
int  rand( )

void srand(seed)
   unsigned int seed;
```

Defined in `rand.c` in `rts.src`

Description Two functions work together to provide pseudo-random sequence generation:

- The **rand** function returns pseudo-random 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 pseudo-random 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.

Syntax

```
#include <stdlib.h>
void *realloc(ptr, size)
   void   *ptr;    /* pointer to object to change    */
   size_t size;    /* new size (in bytes) of packet  */
```

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, then realloc behaves like malloc.
- If `ptr` points to unallocated space, the function takes no action and returns.
- If the space cannot be allocated, the original memory space is not changed and realloc returns 0.
- If `size`=0 and `ptr` is not null, then realloc frees the space that `ptr` points to.

If, in order to allocate more space, the entire object must be moved, realloc returns a pointer to the new space. Any memory freed by this operation is deallocated. If an error occurs, the function returns a null pointer (0).

The memory that realloc uses is in a special memory pool or heap. An assembly language module called `sysmem.asm` defines this memory pool as uninitialized named section called `.sysmem`. The constant `__SYSMEM_SIZE` defines the size of the heap as 2048 words. If necessary, you can change the size of the heap by changing the value of `__SYSMEM_SIZE` and reassembling `sysmem.asm`. For more information, refer to Section 4.1.4, Dynamic Memory Allocation, on page 4-6.

Syntax

```
#include <math.h>

double sin(x)
   double x;
```

Defined in sin.obj in rts.lib

Description The sin function returns the sine of a floating-point number, x. x is an angle expressed in radians. An argument with a large magnitude may produce a result with little or no significance.

Example

```
double radian, sval;  /* sval is returned by sin */

radian = 3.1415927;
sval = sin(radian);   /* -1 is returned by sin   */
```

Syntax

```
#include <math.h>

double sinh(x)
   double x;
```

Defined in `sinh.obj` in `rts.lib`

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 */
```

Syntax

```
#include  <math.h>

double sqrt(x)
   double x;
```

Defined in `sqrt.obj` in `rts.lib`

Description The sqrt function returns the nonnegative square root of a real number `x`. A domain error occurs if the argument is negative.

Example

```
double x, y;

x = 100.0;
y = sqrt(x);            /* return value = 10.0 */
```

Syntax

```
#include <string.h>
char *strcat(s1, s2)
   char *s1, *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.

Syntax

```
#include  <string.h>

char *strchr(s, c)
    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).

Syntax

```
#include <string.h>
int  strcoll(s1, s2)
   char *s1, *s2;

int  strcmp(s1, s2)
   char *s1, *s2;
```

Defined in `strcmp.c` in `rts.src`

Description The strcmp and strcoll functions compare `s2` with `s1`. The functions are equivalent; both functions are supported to provide compatibility with ANSI C.

The 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`.

Syntax

```
#include <string.h>
char *strcpy(s1, s2)
   char *s1, *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`.

Syntax

```
#include <string.h>

size_t strcspn(s1, s2)
   char   *s1, *s2;
```

Defined in `strcspn.c` in `rts.src`

Description The strcspn function returns the length of the initial segment of `s1`, *which is entirely made up of characters that are not in* `s2`. If the first character in `s1` is in `s2`, the function returns 0.

Syntax

```
#include  <string.h>

char *strerror(errnum)
   int  errnum;
```

Defined in `strerror.c` in `rts.src`

Description The strerror function returns the string `"function error"`. This function is supplied to provide ANSI compatibility.

Syntax

```
#include  <time.h>

size_t  *strftime(s, maxsize, format, timeptr)
   char     *s, *format;
   size_t  maxsize;
   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 equivalent of either AM or PM
%S the second as a decimal number (00—50)
%U the week number of the year (Sunday is the first day of the week) as a decimal number (00—52)
%x the date representation
%X the time representation
%y the year without century as a decimal number (00—99)
%Y the year with century as a decimal number
%Z the time zone name, or by no characters if no time zone exists

For more information about the functions and types that the `time.h` header declares, refer to Section 5.1.10 on page 5-8.

Syntax

```
#include <string.h>

size_t strlen(s)
   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.

Syntax

```
#include <string.h>
char *strncat(s1, s2, n)
   char   *s1, *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 the end of `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`.

Syntax

```
#include <string.h>

int  strncmp(s1, s2, n)
   char   *s1, *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`.

Syntax

```
#include <string.h>

char *strncpy(s1, s2, n)
   char   *s1, *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`.

Syntax

```
#include <string.h>
char *strpbrk(s1, s2)
   char *s1, *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).

Syntax

```
#include  <string.h>
char *strrchr(s ,c)
   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).

Syntax

```
#include  <string.h>
size_t *strspn(s1, s2)
   int   *s1, *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.

Syntax

```
#include  <string.h>
char *strstr(s1, s2)
   char *s1, *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, then strstr returns `s1`.

Syntax

```
#include  <stdlib.h>
double strtod(nptr, endptr)
   char  *nptr;
   char  **endptr;

long int strtol(nptr, endptr, base)
   char  *nptr;
   char  **endptr;
   int   base;

unsigned long int strtoul(nptr, endptr, base)
   char  *nptr;
   char  **endptr;
   int   base;
```

Defined in

`strtod.c` in `rts.src`
`strtol.c` in `rts.src`
`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`.

- 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 spacebar, horizontal or vertical tab, carriage return, form feed, or newline. Following the space is an optional *sign*, and then *digits* that represent the integer portion of the number. The fractional part of the number follows, then the exponent, including an optional *sign*.

The first unrecognized character terminates the string. The pointer that `endptr` points to is set to point to this character.

Syntax

```
#include <string.h>
char *strtok(s1, s2)
   char *s1, *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.

Syntax

```
#include <math.h>

double tan(x)
   double x;
```

Defined in `tan.obj` in `rts.lib`

Description The tan function returns the tangent of a floating-point number, `x`. `x` is an angle expressed in radians. An argument with a large magnitude may produce a result with little or no significance.

Example

```
double x, y;

x = 3.1415927/4.0;
y = tan(x);                 /* return value = 1.0 */
```

Syntax

```
#include <math.h>
double tanh(x)
   double x;
```

Defined in `tanh.obj` in `rts.lib`

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 */
```

Syntax

```
#include  <time.h>
time_t  time(timer)
   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 Section 5.1.10 on page 5-8.

Note:

The time function is target-system specific, so you must write your own time function.

Syntax

```
#include <ctype.h>
int  toascii(c)
   char  c;
```

Defined in `toascii.c` in `rts.src`

Description The toascii function ensures that `c` is a valid ASCII character by masking the lower seven bits. There is also an equivalent macro call _toascii.

Syntax

```
#include <ctype.h>
int  tolower(c)
   char  c;

int  toupper(c)
   char  c;
```

Defined in `tolower.c` in `rts.scr`
`toupper.c` in `rts.src`

Description Two functions convert the case of a single alphabetic character, `c`, to upper or lower case:

- The **tolower** function converts an uppercase argument to lowercase. If `c` is 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.

Syntax

```
#include  <stdarg.h>

type    va_arg(ap, type)
        void  va_end(ap)
        void  va_start(ap, parmN)
        va_list  *ap
```

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 run time. The `ap` parameter points to an argument in the variable-argument list.

- The **va_start** macro initializes `ap` to point to the first argument in an argument list for the variable-argument function. The *parmN* parameter points to the rightmost parameter in the fixed, declared list.
- The **va_arg** macro returns the value of the next argument in a call to a variable-argument function. Each time you call va_arg, it modifies `ap` so that successive arguments for the variable-argument function can be returned by successive calls to va_arg (va_arg modifies `ap` to point to the next argument in the list). The *type* parameter is a type name; it is the type of the current argument in the list.
- The **va_end** macro resets the stack environment after va_start and va_arg are used.

Note that you must call va_start to initialize `ap` before calling va_arg or va_end.

Example

```
int    printf(fmt)   /* Has 1 fixed argument and      */

       char  *fmt    /* additional variable arguments  */
{
       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               */
}
```

Appendix A

Error Messages

Compiler error messages are displayed in the following format, which shows the line number in which the error occurs and the text of the message:

*"name***.c", line** *n* **:** *error message*

These types of errors are not fatal.

Fatal Error messages

The errors listed below cause the compiler to abort immediately.

- **>> cannot allocate sufficient memory**

  The compiler requires a minimum of 512K bytes of memory to run; this message indicates that this amount is not available. Supply more dynamic RAM.

- **>> can't open "filename" as source**

  The compiler cannot find the file name as entered. Check for spelling errors and check to see that the named file actually exists.

- **>> can't open "filename" as intermediate file**

  The compiler cannot create the output file. This is usually caused by either an error in the syntax of the filename or a full disk.

- **>> illegal extension "ext" on output file**

  The intermediate file cannot have a ".c" extension.

- **>> fatal errors found: no intermediate file produced**

  This message is printed after an unsuccessful compilation. Correct the errors (other messages will indicate particular errors) and try compilation again.

- **>> cannot recover from earlier errors: aborting**

  An error has occurred that prevents the compiler from continuing.

Appendix B

Preprocessor Directives

The C preprocessor provided with this package is standard and follows Kernighan and Ritchie exactly. This appendix summarizes the directives that the preprocessor supports. Generally, the directives are organized alphabetically, one directive per page; however, related directives (such as #if/#else) are presented together on one page. Here's an alphabetical table of contents for the preprocessor directives reference:

Syntax **#define** *name[(arg,...,arg)] token-string*

#undef *name*

Description The preprocessor supports two directives for defining and undefining macros and constants:

- The **#define** directive assigns a string to a macro. Subsequent occurrences of *name* are replaced by *token-string*. The *name* can be immediately followed by an argument list; the arguments are separated by commas, and the list is enclosed in parentheses. Each occurrence of an argument is replaced by the corresponding set of tokens from the comma-separated string.

  When a macro with arguments is expanded, the arguments are placed into the expanded *token-string* unchanged. After the entire *token-string* is expanded, the preprocessor scans again for names to expand at the beginning of the newly created *token-string*, which allows for nested macros.

  Note that there is no space between *name* and the open parenthesis at the beginning of the argument list. A trailing semicolon is not required; if used, it is treated as part of the *token-string*.

- The **#undef** directive undefines the macro *name*; that is, it causes the preprocessor to forget the definition of *name*.

Example The following example defines the constant `f`:

```
#define f(a,b,c) 3*a+b-c
```

The following line of code uses the definition of `f`:

```
f(27,begin,minus)
```

This line is expanded to:

```
3*27+begin-minus
```

To undefine `f`, enter:

```
#undef  f
```

Syntax

#if *constant-expression*
    code to compile if condition is true
[**#else**
    code to compile if condition is false]
#endif

#ifdef *name*
    code to compile if name is defined
[**#else**
    code to compile if name is not defined]
#endif

#ifndef *name*
    code to compile if name is not defined
[**#else**
    code to compile if name is defined]
#endif

Description The C preprocessor supports five conditional processing directives:

- Three directives can begin a conditional block:
  - The **#if** directive tests an expression. The code following an #if directive (up to an #else or an #endif) is compiled if the *constant-expression* evaluates to a nonzero value. All binary non-assignment C operators, the ?: operator, the unary –, !, and % operators are legal in *constant-expression.* The precedence of the operators is the same as in the definition of the C language. The preprocessor also supports a unary operator named **defined**, which can be used in *constant-expression* in one of two forms:

    1) `defined(name)` **or**

    2) `defined name`

    This allows the the utility of #ifdef and #ifndef in an #if directive. Only these operators, integer constants, and names which are known by the preprocessor should be used in *constant-expression.* In particular, the *sizeof* operator should not be used.
  - The **#ifdef** directive tests to see if *name* is a defined constant. The code following an #ifdef directive (up to an #else or an #endif) is compiled if *name* is defined (by the #define directive) and it has not been undefined by the #undef directive.

  - The **#ifndef** directive tests to see if *name* is *not* a defined constant. The code following an #ifndef directive (up to an #else or an #endif) is compiled if *name* is not defined (by the #define directive) or if it was undefined by the #undef directive.

- The **#else** directive begins an alternate block of code that is compiled if:

  - The condition tested by #if is false.
  - The name tested by #ifdef is not defined.
  - The name tested by #ifndef is defined.

  Note that the #else portion of a conditional block is *optional*; if the #if, #ifdef, or #ifndef test is not successful, then the preprocessor continues with the code following the #endif.

- The **#endif** directive ends a conditional block. Each #if, #ifdef, and #ifndef directive must have a matching #endif. Conditional compilation sequences can be nested.

Syntax

#include *"filename"*

or

#include *<filename>*

Description The #include directive tells the preprocessor to read source statements from another file. The preprocessor includes (at the point in the code where #include is encountered) the contents of the *filename*, which are then processed. You can enclose the *filename* in double quotes or in angle brackets.

The *filename* can be a complete pathname or a filename with no path information.

- If you provide path information for *filename*, the preprocessor uses that path and *does not look* for the file in any other directories.
- If you do not provide path information and you enclose the *filename* in **double quotes**, the preprocessor searches for the file in:
  1) The directory that contains the current source file. (The current source file refers to the file that is being processed when the preprocessor encounters the #include directive.)
  2) Any directories named with the –i preprocessor option.
  3) Any directories named with the C_DIR environment variable.
- If you do not provide path information and you enclose the *filename* in **angle brackets**, the preprocessor searches for the file in:
  1) Any directories named with the –i preprocessor option.
  2) Any directories named with the C_DIR environment variable.

Note:

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

Syntax **#line** *integer-constant ["filename"]*

Description The #line directive generates line control information for the next pass of the compiler. The *integer-constant* is the line number of the next line, and the *filename* is the file where that line exists. If you do not provide a filename, the current filename (specified by the last #line directive) is unchanged.

This directive effectively sets the `__LINE__` and `__FILE__` symbols.

Appendix C

Increasing Code Generation Efficiency

The efficiency of the code generated by the TMS320C30 C compiler depends largely on how effectively you take advantage of the C compiler optimizations. The following list describes the key constructs that can vastly improve the compiler's effectiveness.

- **Use register variables** for often-used variables. This is particularly important for pointer variables (the compiler allocates four registers for pointer register variables). For example, the following code fragment exchanges one memory object with another:

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

  Without register variables, this code takes 12 instructions and 19*n* cycles. With register variables, this code takes only 4 instructions and 7*n* cycles.

- **Avoid integer multiplies** (or use the –m option). The TMS320C30 MPYI instruction uses 24-bit operands, forcing the compiler to use run-time support to do full 32-bit arithmetic. You can use the –m option, which forces the compiler to use MPYI, if you know 24-bit multiplies are sufficient for your application.

- **Pre-compute subexpressions**, especially array references in loops. Assign commonly used expressions to register variables where possible.

- **Use *++ to step through arrays**, rather than using an index to recalculate the address each time through a loop.

As an example of pre-computing subexpressions and using *++ to step through arrays, consider the following loops:

```
main()
  {
     float a[10], b[10];
     int i;

     for (i = 0; i < 10; ++i)
      a[i] = (a[i] * 20) + b[i];
  }
```

Executes in 19 Cycles

```
main()
  {
     float a[10], b[10];
     int i;
     register float *p = a,  *q = b;

     for (i = 0; i < 10; ++i)
      *p++ = (*p * 20) + *q++;
  }
```

Executes in 12 Cycles

- ❑ **Use structure assignments to copy blocks of data**. The compiler generates very efficient code for structure assignments; therefore, nest objects within structures and use simple assignments to copy them.
- ❑ **Avoid large local frames, and declare the most often used local variables first**. The compiler uses indirect addressing with an 8-bit offset to access local data. To access objects on the local frame that have offsets greater than 255, the compiler must first load the offset into an index register resulting in1 extra instruction and 2 cycles of pipeline delay.
- ❑ **Avoid the big memory model.** The big model is inefficient because the compiler reloads the data page pointer (DP) before each access to a global or static variable. If you have large array objects, use `malloc()` to dynamically allocate and access the variables via pointers rather than declaring them globally. For example:

```
int a[100000];
...
a[i] = 10;     /* 11 cycles */
```

Inefficient for Large Array Objects

```
int *a = (int *)malloc(100000); /
...
a[i] = 10;    /* 5 cycles */
```

Efficient for Large Array Objects

Index

C

D

E

F

G

H

I

M

N

O

S

U

V

W

X

TI Worldwide Sales Offices

ALABAMA: Huntsville: 500 Wynn Drive, Suite 514, Huntsville, AL 35805, (205) 837-7530.

ARIZONA: Phoenix: 8825 N. 23rd Ave., Phoenix, AZ 85021, (602) 995-1007;**TUCSON:** 818 W. Miracle Mile, Suite 43, Tucson, AZ 85705, (602) 292-2640.

CALIFORNIA: Irvine: 17891 Cartwright Dr., Irvine, CA 92714, (714) 660-1200; **Roseville:** 1 Sierra Gate Plaza, Roseville, CA 95678, (916) 786-9208; **San Diego:** 4333 View Ridge Ave., Suite 100, San Diego, CA 92123, (619) 278-9601; **Santa Clara:** 5353 Betsy Ross Dr., Santa Clara, CA 95054, (408) 980-9000; **Torrance:** 690 Knox St., Torrance, CA 90502, (213) 217-7010; **Woodland Hills:** 21220 Erwin St., Woodland Hills, CA 91367, (818) 704-7759.

COLORADO: Aurora: 1400 S. Potomac Ave., Suite 101, Aurora, CO 80012, (303) 368-8000.

CONNECTICUT: Wallingford: 9 Barnes Industrial Park Rd., Barnes Industrial Park, Wallingford, CT 06492, (203) 269-0074.

FLORIDA: Altamonte Springs: 370 S. North Lake Blvd, Altamonte Springs, FL 32701, (305) 260-2116; **Ft. Lauderdale:** 2950 N.W. 62nd St., Ft. Lauderdale, FL 33309, (305) 973-8502; **Tampa:** 4803 George Rd., Suite 390, Tampa, FL 33634, (813) 885-7411.

GEORGIA: Norcross: 5515 Spalding Drive, Norcross, GA 30092, (404) 662-7900

ILLINOIS: Arlington Heights: 515 W. Algonquin, Arlington Heights, IL 60005, (312) 640-2925.

INDIANA: Ft. Wayne: 2020 Inwood Dr., Ft. Wayne, IN 46815, (219) 424-5174; **Carmel:** 550 Congressional Dr., Carmel, IN 46032, (317) 573-6400.

IOWA: Cedar Rapids: 373 Collins Rd. NE, Suite 201, Cedar Rapids, IA 52402, (319) 395-9550.

KANSAS: Overland Park: 7300 College Blvd., Lighton Plaza, Overland Park, KS 66210, (913) 451-4511.

MARYLAND: Columbia: 8815 Centre Park Dr., Columbia MD 21045, (301) 964-2003.

MASSACHUSETTS: Waltham: 950 Winter St., Waltham, MA 02154, (617) 895-9100.

MICHIGAN: Farmington Hills: 33737 W. 12 Mile Rd., Farmington Hills, MI 48018, (313) 553-1569. **Grand Rapids:** 3075 Orchard Vista Dr. S.E., Grand Rapids, MI 49506, (616) 957-4200.

MINNESOTA: Eden Prairie: 11000 W. 78th St., Eden Prairie, MN 55344 (612) 828-9300.

MISSOURI: St. Louis: 11816 Borman Drive, St. Louis, MO 63146, (314) 569-7600.

NEW JERSEY: Iselin: 485E U.S. Route 1 South, Parkway Towers, Iselin, NJ 08830 (201) 750-1050.

NEW MEXICO: Albuquerque: 2820-D Broadbent Pkwy NE, Albuquerque, NM 87107, (505) 345-2555.

NEW YORK: East Syracuse: 6365 Collamer Dr., East Syracuse, NY 13057, (315) 463-9291; **Melville:** 1895 Walt Whitman Rd., P.O. Box 2936, Melville, NY 11747, (516) 454-6600; **Pittsford:** 2851 Clover St., Pittsford, NY 14534, (716) 385-6770; **Poughkeepsie:** 385 South Rd., Poughkeepsie, NY 12601, (914) 473-2900.

NORTH CAROLINA: Charlotte: 8 Woodlawn Green, Woodlawn Rd., Charlotte, NC 28210, (704) 527-0933; **Raleigh:** 2809 Highwoods Blvd., Suite 100, Raleigh, NC 27625, (919) 876-2725.

OHIO: Beachwood: 23775 Commerce Park Rd., Beachwood, OH 44122, (216) 464-6100; **Beavercreek:** 4200 Colonel Glenn Hwy., Beavercreek, OH 45431, (513) 427-6200.

OREGON: Beaverton: 6700 SW 105th St., Suite 110, Beaverton, OR 97005, (503) 643-6758.

PENNSYLVANIA: Blue Bell: 670 Sentry Pkwy, Blue Bell, PA 19422, (215) 825-9500.

PUERTO RICO: Hato Rey: Mercantil Plaza Bldg., Suite 505, Hato Rey, PR 00918, (809) 753-8700.

TENNESSEE: Johnson City: Erwin Hwy, P.O. Drawer 1255, Johnson City, TN 37605 (615) 461-2192.

TEXAS: Austin: 12501 Research Blvd., Austin, TX 78759, (512) 250-7655; **Richardson:** 1001 E. Campbell Rd., Richardson, TX 75081, (214) 680-5082; **Houston:** 9100 Southwest Frwy., Suite 250, Houston, TX 77074, (713) 778-6592; **San Antonio:** 1000 Central Parkway South, San Antonio, TX 78232, (512) 496-1779.

UTAH: Murray: 5201 South Green St., Suite 200, Murray, UT 84123, (801) 266-8972.

WASHINGTON: Redmond: 5010 148th NE, Bldg B, Suite 107, Redmond, WA 98052, (206) 881-3080.

WISCONSIN: Brookfield: 450 N. Sunny Slope, Suite 150, Brookfield, WI 53005, (414) 782-2899.

CANADA: Nepean: 301 Moodie Drive, Mallorn Center, Nepean, Ontario, Canada, K2H9C4, (613) 726-1970. **Richmond Hill:** 280 Centre St. E., Richmond Hill L4C1B1, Ontario, Canada (416) 884-9181; **St. Laurent:** Ville St. Laurent Quebec, 9460 Trans Canada Hwy., St. Laurent, Quebec, Canada H4S1R7, (514) 336-1860.

ARGENTINA: Texas Instruments Argentina Viamonte 1119, 1053 Capital Federal, Buenos Aires, Argentina, 541/748-3699

AUSTRALIA (& NEW ZEALAND): Texas Instruments Australia Ltd.: 6-10 Talavera Rd., North Ryde (Sydney), New South Wales, Australia 2113, 2 + 887-1122; 5th Floor, 418 St. Kilda Road, Melbourne, Victoria, Australia 3004, 3 + 267-4677; 171 Philip Highway, Elizabeth, South Australia 5112, 8 + 255-2066.

AUSTRIA: Texas Instruments Ges.m.b.H.: Industriestrabe B/16, A-2345 Brunn/Gebirge, 2236-846210.

BELGIUM: Texas Instruments N.V. Belgium S.A.: 11, Avenue Jules Bondetlaan 11, 1140 Brussels, Belgium, (02) 242-3080.

BRAZIL: Texas Instruments Electronicos do Brasil Ltda.: Rua Paes Leme, 524-7 Andar Pinheiros, 05424 Sao Paulo, Brazil, 0815-6166.

DENMARK: Texas Instruments A/S, Mairelundvej 46E, 2730 Herlev, Denmark, 2 - 91 74 00.

FINLAND: Texas Instruments Finland OY: Ahertajantie 3, P.O. Box 81, ESPOO, Finland, (90) 0-461-422.

FRANCE: Texas Instruments France: Paris Office, BP 67 8-10 Avenue Morane-Saulnier, 78141 Velizy-Villacoublay cedex (1) 30 70 1003.

GERMANY (Fed. Republic of Germany): Texas Instruments Deutschland GmbH: Haggertystrasse 1, 8050 Freising, 8161 + 80-4591; Kurfuerstendamm 195/196, 1000 Berlin 15, 30 + 882-7365; III, Hagen 43/Kibbelstrasse, .19, 4300 Essen, 201-24250; Kirchhorsterstrasse 2, 3000 Hannover 51, 511 + 648021; Maybachstrabe 11, 7302 Ostfildern 2-Nelingen, 711 + 34030.

HONG KONG: Texas Instruments Hong Kong Ltd., 8th Floor, World Shipping Ctr., 7 Canton Rd., Kowloon, Hong Kong, (852) 3-7351223.

IRELAND: Texas Instruments (Ireland) Limited: 7/8 Harcourt Street, Stillorgan, County Dublin, Eire, 1 781677.

ITALY: Texas Instruments Italia S.p.A. Divisione Semiconduttori: Viale Europa, 40, 20093 Cologne Monzese (Milano), (02) 253001; Via Castello della Magliana, 38, 00148 Roma, (06) 5222651; Via Amendola, 17, 40100 Bologna, (051) 554004.

JAPAN: Tokyo Marketing/Sales (Headquarters): Texas Instruments Japan Ltd., MS Shibaura Bldg., 9F, 4-13-23 Shibaura, Minato-ku, Tokyo 108, Japan, 03-769-8700. Texas Instruments Japan Ltd.: Nissho-Iwai Bldg. 5F, 30 Imabashi 3-chome, Higashi-ku, Osaka 541, Japan, 06-294-1881; Daini Toyota West Bldg. 7F, 10-27 Meieki 4-chome, Nakamura-ku, Nagoya 450, 052-583-8691; Daiichi Seimei Bldg. 6F, 3-10 Oyama-cho, Kanazawa 920, Ishikawa-ken, 0762-23-5471; Daiichi Olympic Tachikawa Bldg. 6F, 1-25-12 Akebono-cho, Tachikawa 190, Tokyo, 0425-27-6426; Matsumoto Showa Bldg. 6F, 2-11 Fukashi 1-chome, Matsumoto 390, Nagano-ken, 0263-33-1060; Yokohama Nishiguchi KN Bldg. 6F, 2-8-4 Kita-Saiwai-cho, Nishi-ku, Yokohama 220, 045-322-6741; Nihon Seimei Kyoto Yasaka Bldg. 5F, 843-2 Higashi Shiokohjidori, Nishinotoh-in Higashi-iru, Shiokouji, Shimogyo-ku, Kyoto 600, 075-341-7713; 2597-1, Aza Harudai, Oaza Yasaka, Kitsuki 873, Oita-ken, 09786-3-3211; Miho Plant, 2350 Kihara Miho-mura, Inashiki-gun 300-04, Ibaragi-ken, 0298-85-2541.

KOREA: Texas Instruments Korea Ltd., 28th Fl., Trade Tower, #159, Samsung-Dong, Kangnam-ku, Seoul, Korea 2 + 551-2810.

MEXICO: Texas Instruments de Mexico S.A.: Alfonso Reyes—115, Col. Hipodromo Condesa, Mexico, D.F., Mexico 06120, 525/525-3860.

MIDDLE EAST: Texas Instruments: No. 13, 1st Floor Mannai Bldg., Diplomatic Area, P.O. Box 26335, Manama Bahrain, Arabian Gulf, 973 + 274681.

NETHERLANDS: Texas Instruments Holland B.V., 19 Hogehilweg, 1100 AZ Amsterdam—Zuidoost, Holland 20 + 5602911.

NORWAY: Texas Instruments Norway A/S: PB106, Refstad 0585, Oslo 5, Norway, (2) 155090.

PEOPLES REPUBLIC OF CHINA: Texas Instruments China Inc., Beijing Representative Office, 7-05 Citic Bldg., 19 Jianguomenwai Dajje, Beijing, China, (861) 5002255, Ext. 3750.

PHILIPPINES: Texas Instruments Asia Ltd.: 14th Floor, Ba- Lepanto Bldg., Paseo de Roxas, Makati, Metro Manila, Philippines, 817-60-31.

PORTUGAL: Texas Instruments Equipamento Electronico (Portugal), Lda.: Rua Eng. Frederico Ulrich, 2650 Moreira Da Maia, 4470 Maia, Portugal, 2-948-1003.

SINGAPORE (+ INDIA, INDONESIA, MALAYSIA, THAILAND): Texas Instruments Singapore (PTE) Ltd., Asia Pacific Division, 101 Thompson Rd. #23-01, United Square, Singapore 1130, 350-8100.

SPAIN: Texas Instruments Espana, S.A.: C/Jose Lazaro Galdiano No. 6, Madrid 28036, 1/458.14.58.

SWEDEN: Texas Instruments International Trade Corporation (Sverigefilialen): S-164-93, Stockholm, Sweden, 8 - 752-5800.

SWITZERLAND: Texas Instruments, Inc., Reidstrasse 6, CH-8953 Dietikon (Zuerich) Switzerland, 1-740 2220.

TAIWAN: Texas Instruments Supply Co., 9th Floor Bank Tower, 205 Tun Hwa N. Rd., Taipei, Taiwan, Republic of China, 2 + 713-9311.

UNITED KINGDOM: Texas Instruments Limited: Manton Lane, Bedford, MK41 7PA, England, 0234 270111.

TEXAS INSTRUMENTS

TI Sales Offices

ALABAMA: Huntsville (205) 837-7530.

ARIZONA: Phoenix (602) 995-1007;
Tucson (602) 292-2640.

CALIFORNIA: Irvine (714) 660-1200;
Roseville (916) 786-9208;
San Diego (619) 278-9601;
Santa Clara (408) 980-9000;
Torrance (213) 217-7010;
Woodland Hills (818) 704-7759.

COLORADO: Aurora (303) 368-8000.

CONNECTICUT: Wallingford (203) 269-0074.

FLORIDA: Altamonte Springs (305) 260-2116;
Ft. Lauderdale (305) 973-8502;
Tampa (813) 885-7411.

GEORGIA: Norcross (404) 662-7900.

ILLINOIS: Arlington Heights (312) 640-2925.

INDIANA: Carmel (317) 573-6400;
Ft. Wayne (219) 424-5174.

IOWA: Cedar Rapids (319) 395-9550.

KANSAS: Overland Park (913) 451-4511.

MARYLAND: Columbia (301) 964-2003.

MASSACHUSETTS: Waltham (617) 895-9100.

MICHIGAN: Farmington Hills (313) 553-1569;
Grand Rapids (616) 957-4200.

MINNESOTA: Eden Prairie (612) 828-9300.

MISSOURI: St. Louis (314) 569-7600.

NEW JERSEY: Iselin (201) 750-1050.

NEW MEXICO: Albuquerque (505) 345-2555.

NEW YORK: East Syracuse (315) 463-9291;
Melville (516) 454-6600;
Pittsford (716) 385-6770;
Poughkeepsie (914) 473-2900.

NORTH CAROLINA: Charlotte (704) 527-0933;
Raleigh (919) 876-2725.

OHIO: Beachwood (216) 464-6100;
Beaver Creek (513) 427-6200.

OREGON: Beaverton (503) 643-6758.

PENNSYLVANIA: Blue Bell (215) 825-9500.

PUERTO RICO: Hato Rey (809) 753-8700.

TENNESSEE: Johnson City (615) 461-2192.

TEXAS: Austin (512) 250-7655;
Houston (713) 778-6592;
Richardson (214) 680-5082;
San Antonio (512) 496-1779.

UTAH: Murray (801) 266-8972.

WASHINGTON: Redmond (206) 881-3080.

WISCONSIN: Brookfield (414) 782-2899.

CANADA: Nepean, Ontario (613) 726-1970;
Richmond Hill, Ontario (416) 884-9181;
St. Laurent, Quebec (514) 336-1860.

TI Regional Technology Centers

CALIFORNIA: Irvine (714) 660-8105;
Santa Clara (408) 748-2220;

GEORGIA: Norcross (404) 662-7945.

ILLINOIS Arlington Heights (312) 640-2909.

MASSACHUSETTS: Waltham (617) 895-9196.

TEXAS: Richardson (214) 680-5066.

CANADA: Nepean, Ontario (613) 726-1970.

TI Distributors

TI AUTHORIZED DISTRIBUTORS
Arrow/Kierulff Electronics Group
Arrow (Canada)
Future Electronics (Canada)
GRS Electronics Co., Inc.
Hall-Mark Electronics
Marshall Industries
Newark Electronics
Schweber Electronics
Time Electronics
Wyle Laboratories
Zeus Components

–OBSOLETE PRODUCT ONLY–
Rochester Electronics, Inc.
Newburyport, Massachusetts
(508) 462-9332

ALABAMA: Arrow/Kierulff (205) 837-6955;
Hall-Mark (205) 837-8700; Marshall (205) 881-9235;
Schweber (205) 895-0480.

ARIZONA: Arrow/Kierulff (602) 437-0750;
Hall-Mark (602) 437-1200; Marshall (602) 496-0290;
Schweber (602) 431-0030; Wyle (602) 866-2888.

CALIFORNIA: **Los Angeles/Orange County:**
Arrow/Kierulff (818) 701-7500, (714) 838-5422;
Hall-Mark (818) 773-4500, (714) 669-4100;
Marshall (818) 407-0101, (818) 459-5500,
(714) 458-5395; Schweber (818) 880-9686;
(714) 863-0200, (213) 320-8090; Wyle (818) 880-9000,
(714) 863-9953; Zeus (714) 921-9000; (818) 889-3838;
Sacramento: Hall-Mark (916) 624-9781;
Marshall (916) 635-9700; Schweber (916) 364-0222;
Wyle (916) 638-5282;
San Diego: Arrow/Kierulff (619) 565-4800;
Hall-Mark (619) 268-1201; Marshall (619) 578-9600;
Schweber (619) 450-0454; Wyle (619) 565-9171;
San Francisco Bay Area: Arrow/Kierulff (408) 745-6600,
Hall-Mark (408) 432-0900; Marshall (408) 942-4600;
Schweber (408) 432-7171; Wyle (408) 727-2500;
Zeus (408) 998-5121.

COLORADO: Arrow/Kierulff (303) 790-4444;
Hall-Mark (303) 790-1662; Marshall (303) 451-8383;
Schweber (303) 799-0258; Wyle (303) 457-9953.

CONNETICUT: Arrow/Kierulff (203) 265-7741;
Hall-Mark (203) 271-2844; Marshall (203) 265-3822;
Schweber (203) 264-4700.

FLORIDA: **Ft. Lauderdale:**
Arrow/Kierulff (305) 429-8200; Hall-Mark (305) 971-9280;
Marshall (305) 977-4880; Schweber (305) 977-7511;
Orlando: Arrow/Kierulff (407) 323-0252;
Hall-Mark (407) 830-5855; Marshall (407) 767-8585;
Schweber (407) 331-7555; Zeus (407) 365-3000;
Tampa: Hall-Mark (813) 530-4543;
Marshall (813) 576-1399; Schweber (813) 541-5100.

GEORGIA: Arrow/Kierulff (404) 449-8252;
Hall-Mark (404) 447-8000; Marshall (404) 923-5750;
Schweber (404) 449-9170.

ILLINOIS: Arrow/Kierulff (312) 250-0500;
Hall-Mark (312) 860-3800; Marshall (312) 490-0155;
Newark (312) 784-5100; Schweber (312) 364-3750.

INDIANA: **Indianapolis:** Arrow/Kierulff (317) 243-9353;
Hall-Mark (317) 872-8875; Marshall (317) 297-0483;
Schweber (317) 843-1050.

IOWA: Arrow/Kierulff (319) 395-7230;
Schweber (319) 373-1417.

KANSAS: **Kansas City:** Arrow/Kierulff (913) 541-9542;
Hall-Mark (913) 888-4747; Marshall (913) 492-3121;
Schweber (913) 492-2922.

TEXAS
INSTRUMENTS

MARYLAND: Arrow/Kierulff (301) 995-6002;
Hall-Mark (301) 988-9800; Marshall (301) 235-9464;
Schweber (301) 840-5900; Zeus (301) 997-1118.

MASSACHUSETTS Arrow/Kierulff (508) 658-0900;
Hall-Mark (508) 667-0902; Marshall (508) 658-0810;
Schweber (617) 275-5100; Time (617) 532-6200;
Wyle (617) 273-7300; Zeus (617) 863-8800.

MICHIGAN: **Detroit:** Arrow/Kierulff (313) 462-2290;
Hall-Mark (313) 462-1205; Marshall (313) 525-5850;
Newark (313) 967-0600; Schweber (313) 525-8100;
Grand Rapids: Arrow/Kierulff (616) 243-0912.

MINNESOTA: Arrow/Kierulff (612) 830-1800;
Hall-Mark (612) 941-2600; Marshall (612) 559-2211;
Schweber (612) 941-5280.

MISSOURI: **St. Louis:** Arrow/Kierulff (314) 567-6888;
Hall-Mark (314) 291-5350; Marshall (314) 291-4650;
Schweber (314) 739-0526.

NEW HAMPSHIRE: Arrow/Kierulff (603) 668-6968;
Schweber (603) 625-2250.

NEW JERSEY: Arrow/Kierulff (201) 538-0900,
(609) 596-8000; GRS Electronics (609) 964-8560;
Hall-Mark (201) 575-4415, (201) 882-9773,
(609) 235-1900; Marshall (201) 882-0320,
(609) 234-9100; Schweber (201) 227-7880.

NEW MEXICO: Arrow/Kierulff (505) 243-4566.

NEW YORK: **Long Island:**
Arrow/Kierulff (516) 231-1009; Hall-Mark (516) 737-0600;
Marshall (516) 273-2424; Schweber (516) 334-7474;
Zeus (914) 937-7400;
Rochester: Arrow/Kierulff (716) 427-0300;
Hall-Mark (716) 425-3300; Marshall (716) 235-7620;
Schweber (716) 424-2222;
Syracuse: Marshall (607) 798-1611.

NORTH CAROLINA: Arrow/Kierulff (919) 876-3132,
(919) 725-8711; Hall-Mark (919) 872-0712;
Marshall (919) 878-9882; Schweber (919) 876-0000.

OHIO: **Cleveland:** Arrow/Kierulff (216) 248-3990;
Hall-Mark (216) 349-4632; Marshall (216) 248-1788;
Schweber (216) 464-2970;
Columbus: Hall-Mark (614) 888-3313;
Dayton: Arrow/Kierulff (513) 435-5563;
Marshall (513) 898-4480; Schweber (513) 439-1800.

OKLAHOMA: Arrow/Kierulff (918) 252-7537;
Schweber (918) 622-8003.

OREGON: Arrow/Kierulff (503) 645-6456;
Marshall (503) 644-5050; Wyle (503) 640-6000.

PENNSYLVANIA: Arrow/Kierulff (412) 856-7000,
(215) 928-1800; GRS Electronics (215) 922-7037;
Marshall (412) 963-0441; Schweber (215) 441-0600,
(412) 963-6804.

TEXAS: **Austin:** Arrow/Kierulff (512) 835-4180;
Hall-Mark (512) 258-8848; Marshall (512) 837-1991;
Schweber (512) 339-0088; Wyle (512) 834-9957;
Dallas: Arrow/Kierulff (214) 380-6464;
Hall-Mark (214) 553-4300; Marshall (214) 233-5200;
Schweber (214) 661-5010; Wyle (214) 235-9953;
Zeus (214) 783-7010;
El Paso: Marshall (915) 593-0706;
Houston: Arrow/Kierulff (713) 530-4700;
Hall-Mark (713) 781-6100; Marshall (713) 895-9200;
Schweber (713) 784-3600; Wyle (713) 879-9953.

UTAH: Arrow/Kierulff (801) 973-6913;
Hall-Mark (801) 972-1008; Marshall (801) 485-1551;
Wyle (801) 974-9953.

WASHINGTON: Arrow/Kierulff (206) 575-4420;
Marshall (206) 486-5747; Wyle (206) 881-1150.

WISCONSIN: Arrow/Kierulff (414) 792-0150;
Hall-Mark (414) 797-7844; Marshall (414) 797-8400;
Schweber (414) 784-9020.

CANADA: **Calgary:** Future (403) 235-5325;
Edmonton: Future (403) 438-2858;
Montreal: Arrow Canada (514) 735-5511;
Future (514) 694-7710;
Ottawa: Arrow Canada (613) 226-6903;
Future (613) 820-8313;
Quebec City: Arrow Canada (418) 871-7500;
Toronto: Arrow Canada (416) 672-7769;
Future (416) 638-4771; Marshall (416) 674-2161;
Vancouver: Arrow Canada (604) 291-2986;
Future (604) 294-1166.

Customer Response Center

TOLL FREE: (800) 232-3200

OUTSIDE USA: (214) 995-6611
(8:00 a.m. – 5:00 p.m. CST)

TEXAS INSTRUMENTS

Printed in U.S.A., July 1989
1604910-9707

SPRU034A