Texas Instruments

TMS320C25 C Compiler

Reference Guide

1989

Digital Signal Processor Products

Reference Guide

TMS320C25 C Compiler

1989

TMS320C25 C Compiler Reference Guide

Texas Instruments

Preface

Read This First

How to Use This Manual

This document contains the following chapters:

Chapter 1	**Introduction and Installation** Provides an overview of the TMS320C25 development tools, describes the code development process, and contains instructions for installing the C compiler.
Chapter 2	**Compiler Operation** Describes the three major components of the C compiler (preprocessor, parser, and code generator), contains instructions for invoking individually each of these components or for invoking a shell program to compile and assemble a C source file. It also discusses linking and archiving C programs.
Chapter 3	**TMS320C25 C Language** Discusses the differences between the C language supported by the TMS320C25 C compiler and standard Kernighan and Ritchie C.
Chapter 4	**Runtime Environment** Contains technical information on how the compiler uses the TMS320C25 architecture; discusses memory and register conventions, stack organization, function-call conventions, and system initialization; provides information needed for interfacing assembly language to C programs.
Chapter 5	**Runtime-Support Functions** Describes the header files that are shipped 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	**Fatal Errors** Shows the format of compiler error messages and lists all the error messages issued by fatal errors.
Appendix B	**Preprocessor Directives** Describes the standard preprocessor directives that the compiler supports.

Related Documentation

You should obtain a copy of ***The C Programming Language*** (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:

- Kochan, Steve G. ***Programming in C***, Hayden Book Company.
- Sobelman, Gerald E. and David E. Krekelberg. ***Advanced C:Techniques and Applications***, Que Corporation, 1985.

The following books, which describe the TMS320C25 and related support tools, are available from Texas Instruments. To obtain TI literature, please call the Texas Instruments Customer Response Center (CRC) at 1–800–232–3200.

- The ***TMS320 Family-Second Generation User's Guide*** (literature number SPRU014) discusses hardware aspects of the TMS320 family second-generation devices, including the TMS320C25. Topics in this user's guide include pin functions, architecture, stack operation, and interfaces; the manual also includes the TMS320C2x assembly language instruction set.
- The ***TMS320 Family-Second Generation Data Sheet*** (literature number SPRS010) contains the recommended operating conditions, electrical specifications, and timing characteristics for the TMS32020 and TMS320C25.
- The ***TMS320C1x/TMS320C2x Assembly Language Tools User's-Guide*** (literature number SPRU018) describes the assembly language tools (assembler, linker, archiver, and object format converter), assembler directives, macros, common object file format, and symbolic debugging directives.

Style and Symbol Conventions

This document uses the following conventions:

- In this document, program listings or examples, interactive displays, filenames, file contents, and symbol names are shown in a `special font`. Examples may use a **`bold version`** of the special font for emphasis. Here is a sample declaration:

```
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.

- Square brackets (**[** and **]**) identify optional information. If you use an option, you specify the information within the brackets; you don't enter the brackets themselves. Here is an example of the syntax to invoke a program:

  dspcc *input file* [*output file*] [*options*]

  The **dspcc** program can use three inputs. The first input, *input file*, is required. The second input *output file*, is optional. If you don't specify an output filename the program uses the input filename with a different extension. The third input is optional and consists of a letter preceded with a dash, that specifies the running mode.

  Square brackets are also used as part of the pathname specification for VMS pathnames; in this case, the brackets are actually part of the pathname (they don't represent options).

Information about Cautions

This book may contain cautions. A **caution** describes a situation that could potentially damage your software or equipment.

This is what a caution looks like.

CAUTION

Contents

1 **Introduction and Installation** **1-1**
- 1.1 Software Development Tools Overview 1-2
- 1.2 Software Installation 1-4
  - 1.2.1 IBM PC with PC-DOS or OS/2 1-4
  - 1.2.2 VAX with VMS 1-5
  - 1.2.3 VAX with Ultrix, Sun-3 with Sun-OS 1-6
  - 1.2.4 Macintosh with MPW 1-7
- 1.3 Getting Started 1-8

2 **C Compiler Operation** **2-1**
- 2.1 Compiling and Assembling a Program 2-2
- 2.2 Invoking the Compiler Tools Individually 2-5
  - 2.2.1 Preprocessing C Code 2-5
  - 2.2.2 Parsing C Code 2-11
  - 2.2.3 Generating Assembly Language Code 2-12
- 2.3 Linking a C Program 2-15
  - 2.3.1 Runtime Initialization and Runtime Support 2-15
  - 2.3.2 Sample Linker Command File 2-16
  - 2.3.3 Autoinitialization (ROM and RAM Models) 2-17
  - 2.3.4 The –c and –cr Linker Options 2-19
- 2.4 Archiving a C Program 2-20

3 **TMS320C25 C Language** **3-1**
- 3.1 Identifiers, Keywords, and Constants 3-2
- 3.2 Data Types 3-4
- 3.3 Object Alignment 3-6
- 3.4 Expressions 3-6
- 3.5 Declarations 3-6
- 3.6 Initialization of Static and Global Variables 3-9
- 3.7 asm Statement (Inline Assembly Language) 3-9
- 3.8 Lexical Scope Rules 3-10

4 **Runtime Environment** **4-1**
- 4.1 Memory Model 4-2

4.1.1 Sections 4-2
4.1.2 Allocating Memory for Static and Global Variables 4-3
4.1.3 RAM and ROM Models 4-3
4.1.4 Managing the Runtime Stack 4-4
4.1.5 Dynamic Memory Allocation 4-5
4.1.6 Packing Structures and Manipulating Fields 4-5
4.2 Register Conventions 4-6
4.2.1 Dedicated Registers 4-6
4.2.2 Using Registers 4-6
4.2.3 Register Variables 4-7
4.3 Function Call Conventions 4-8
4.3.1 Passing Parameters to a Function 4-8
4.3.2 Local Frame Generation 4-8
4.3.3 Function Termination 4-9
4.4 Interfacing C with Assembly Language 4-11
4.4.1 Assembly Language Modules 4-11
4.4.2 Inline Assembly Language 4-14
4.4.3 Modifying Compiler Output 4-15
4.5 Interrupt Handling 4-16
4.6 Integer Expression Analysis 4-18
4.6.1 Arithmetic Overflow and Underflow 4-18
4.6.2 Integer Division and Modulus 4-18
4.7 Floating-Point Expression Analysis 4-19
4.8 System Initialization 4-20
4.8.1 Runtime Stack 4-20
4.8.2 Autoinitialization 4-21

5 Runtime-Support Functions 5-1
5.1 Header Files 5-2
5.1.1 Diagnostic Messages (assert.h) 5-2
5.1.2 Character Typing and Conversion (ctype.h) 5-3
5.1.3 Limits (float.h and limits.h) 5-3
5.1.4 Floating-Point Math (math.h) 5-5
5.1.5 Variable Arguments (stdarg.h) 5-6
5.1.6 Standard Definitions (stddef.h) 5-6
5.1.7 General Utilities (stdlib.h) 5-7
5.1.8 String Functions (string.h) 5-7
5.2 Summary of Runtime-Support Functions and Macros 5-8
5.3 Functions Reference 5-14

A Fatal Errors A-1

B C Preprocessor Directives B-1

Figures

1–1 TMS320C25 Software Development Flow 1-2
2–1 Compiling a C Program 2-5
2–2 Input and Output Files for the C Preprocessor 2-6
2–3 Input and Output Files for the C Parser 2-11
2–4 Input and Output Files for the C Code Generator 2-13
2–5 An Example of a Linker Command File 2-16
2–6 ROM Model of Autoinitialization 2-18
2–7 RAM Model of Autoinitialization 2-19
4–1 An Assembly Language Function 4-12
(a) C program 4-12
(b) Assembly language program 4-13
4–2 Accessing from C a Variable Defined in .bss 4-13
(a) Assembly language program 4-13
(b) C program 4-14
4–3 Accessing from C a Variable not Defined in .bss 4-14
(a) Assembly language program 4-14
(b) C program 4-14

Tables

5–1 Macros That Supply Limits for Characters and Integers 5-4
5–2 Macros That Supply Limits for Floating-Point Numbers 5-5
5–3 Error Message Macro 5-8
5–4 Character Typing and Conversion Functions 5-8
5–5 Floating-Point Math Functions 5-9
5–6 Variable Argument Macros 5-10
5–7 General Utilities 5-11
5–8 String Functions 5-12

Examples

2–1 Method 1 – Invoking Each Tool Individually 2-2
2–2 Method 2 – Using the Shell Program 2-4

Chapter 1

Introduction and Installation

The TMS320C25 is a high-performance CMOS microprocessor, optimized for digital signal processing applications. The TMS320C25 is a member of the second generation of the TMS320 family of digital signal processors.

The TMS320C25 is fully supported by a complete set of hardware and software development tools (Section1.1 describes these tools) including

❑ a C compiler,

❑ an assembler, a linker, and archiver,

❑ a full-speed emulator,

❑ a software simulator, and

❑ a PC-resident software development system.

This reference guide describes the details and characteristics of the TMS320C25 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), as a supplement to this reference guide.

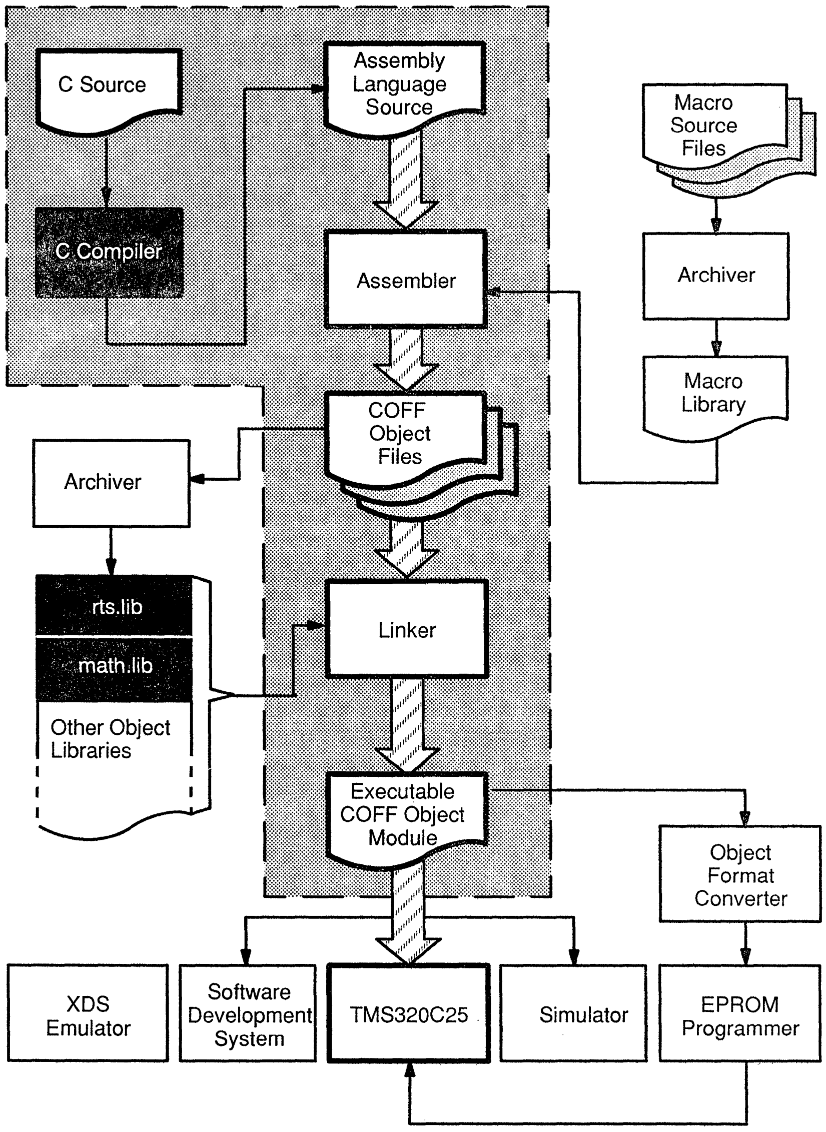
Topics in this introductory chapter include:

Section **Page**
1.1 Software Development Tools Overview 1-2
1.2 Software Installation 1-4
1.3 Getting Started 1-8

1.1 Software Development Tools Overview

Figure 1–1 illustrates the TMS320C25 software development flow. The center portion of the figure highlights the most common path of software development; the other portions are optional.

Figure 1–1. TMS320C25 Software Development Flow



The following list describes the tools that are shown in Figure 1–1. Chapter 2, *C Compiler Operation*, contains instructions for compiling, assembling, linking, and archiving C programs.

- The **C compiler** accepts C source code and produces TMS320C25 assembly language source code. The C compiler has three parts: a preprocessor, a parser, and a code generator. Section 3 describes compiler 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 called a *library*.) It also allows you to modify a library by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is to build a library of object modules. Two object libraries are shipped with the C compiler:
  - `flib.lib` contains floating-point arithmetic routines.
  - `rts.lib` contains standard runtime-support functions and compiler-utility functions.

  These functions and routines 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 library members 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 **TMS320C25 target system** . You can use one of several debugging tools to refine and correct your code; available products include:
  - A **simulator**,
  - An **extended development system** (XDS) emulator, and
  - A **software development system** (SWDS).
- An **object format converter** is also available; it converts a COFF object file into an Intel, Tektronix, or TI-tagged object format file that can be downloaded to an EPROM programmer.

1.2 Software Installation

This section contains step-by-step instructions for installing the TMS320C25 C compiler on the following systems:

- ❑ **IBM PC**
  - ■ PC-DOS (versions 3.x and up)
  - ■ OS/2
- ❑ **Digital Equipment Corporation VAX-11**
  - ■ VMS operating system
  - ■ Ultrix operating system
- ❑ **Sun-3 Workstation**
  - ■ Sun-OS (versions 4.x)
- ❑ **Macintosh**
  - ■ MPW

Note:

To use the TMS320C25 C compiler, you must also have version 5.0 (or later) of the TMS320C25 assembler and linker.

1.2.1 IBM PC with PC-DOS or OS/2

The C compiler package is shipped on double-sided, dual-density diskettes. The compiler requires 512K bytes of available 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 system diskette should be in drive B. The instructions use these symbols for drive names:

A:	Floppy diskette drive for hard disk systems; source drive for dual-drive systems.
B:	Destination or system diskette 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 into drive B the diskette that will contain the tools .

   ❑ On *hard-disk* systems, enter

```
MD C:\DSPTOOLS ↵
```

- On *dual-drive* systems, enter

  ```
  MD B:\DSPTOOLS ↵
  ```

3) Copy the C compiler package onto the hard disk or formatted diskette. Put the product diskette in drive A; if you're using a dual-drive system, put the diskette that will contain the tools into drive B.

   - On *hard-disk* systems, enter

     ```
     COPY  A:\*.* C:\DSPTOOLS\*.* ↵
     ```

   - On *dual-drive* systems, enter

     ```
     COPY  A:\*.* B:\DSPTOOLS\*. ↵
     ```

4) Put the directory that contains the C tools into your system path.

   - On *hard-disk* systems, enter

     ```
     PATH C:\DSPTOOLS; ↵
     ```

   - On *dual-drive* systems, enter

     ```
     PATH B:\DSPTOOLS; ↵
     ```

1.2.2 VAX with VMS

The TMS320C25 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 must 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 and *`DIRECTORY`* with the actual name of the directory.

```
$ allocate           TAPE:
$ mount/for/den=1600TAPE:
$ backup TAPE:320.bck/SELECT=[MASTER.DSPC...] 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 displays the defined symbols on the screen.

1.2.3 VAX with Ultrix, Sun-3 with Sun-OS

The TMS320C25 C compiler product tape was made at 1600 BPI, using the TAR command. Follow these instructions to install the compiler:

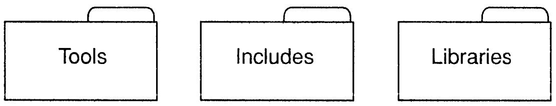
1) Mount the tape on your tape drive.
2) Set the directory where you will store the tools.
3) Enter the TAR command for your system; for example,

   ```
   TAR x
   ```

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

1.2.4 Macintosh with MPW

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

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.

1.3 Getting Started

The TMS320C25 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. The simplest way to compile and assemble a C program is to use the shell program that is shipped 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)*/
/********************************/
int abs(i)
int i;
{
 register int temp = i;
 if (temp < 0) temp *= -1;
 return (temp);
}
```

2) To invoke the shell program to compile and assemble `function.c`; enter:

   dspcl `function` ↵

   The shell program prints the following information as it compiles the program:

```
dspcl function
[function}
C Pre-Processor     Version 5.10
(c) Copyright 1987, 1989, Texas Instruments Incorporated
DSP C Compiler      Version 5.10
(c) Copyright 1987, 1989, Texas Instruments Incorporated
 "function.c": ==> abs
DSP C Codegen       Version 5.10
(c) Copyright 1987, 1989, Texas Instruments Incorporated
 "function.c": ==> abs
DSP COFF Assembler  Version 5.10
(c) Copyright 1987, 1989, Texas Instruments Incorporated
 PASS 1
 PASS 2

No Errors, No Warnings
```

The shell program compiles and assembles `function.c` by invoking the following:

dspcpp	→	C Preprocessor
dspcc	→	C Parser
dspcg	→	Code Generator
dspa	→	Assembler

In this example, `function.c` is the input source file. Do not specify an extension for the input file; the batch file assumes that the input file has a .c extension.

Each tool creates a file that the next tool uses for input; the tools use the filename of the source file (without the .c extension) to name the files that they create. This example uses and creates the following files:

a) The source file `function.c` is input for the preprocessor; the preprocessor creates a modified C source file called `function.cpp`.

b) `function.cpp` is input for the parser; the parser creates an intermediate file called `function.if`.

c) `function.if` is input for the code generator; the code generator creates an assembly language file called `function.asm`.

d) `function.asm` is input for the assembler; the assembler creates an object file called `function.obj`.

3) The final output is an object file. This example creates an object file called `function.obj`. To create an executable object module, link the object file created by the batch file with the runtime-support library `rts.lib`:

```
dsplnk  -c function -o function.out -l rts.lib
```

This example uses the –c linker option because the code was originally from a C program. It uses the –o option to name the output module `function.out`. If you don't use the –o option, the linker names the output module `a.out`.

You can find more information about invoking the compiler tools, the assembly language tools, and the batch files in the following sections:

Section		Page
2.1	Compiling and Assembling a Program	2-2
2.2	Invoking the Compiler Tools Individually	2-5
2.3	Linking a C Program	2-15
2.4	Archiving a C Program	2-20

Chapter 2

C Compiler Operation

The TMS320C25 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 TMS320C25 assembler and linker.

If you choose to run the three compiler steps individually, Section 2.2 describes how to do so.

Topics in this chapter include

Section **Page**

2.1 Compiling and Assembling a Program 2-2
2.2 Invoking the Compiler Tools Individually 2-5
2.3 Linking a C Program 2-15
2.4 Archiving a C Program 2-20

2.1 Compiling and Assembling a Program

The compiler creates a single assembly language source file as output. You can assemble and link this file to form an executable object module. You can compile several C source programs, assemble each of them, and then link them together. (The *TMS320C1x/TMS32C2x Assembly Language Tools User's Guide* describes the TMS320C25 assembler and linker.)

Example 2–1 and Example 2–2 show two different methods for compiling and assembling a C program. Both of these examples compile and assemble a C source file called `program.c` and create an object file called `program.obj`. Example 2–1 shows how you can accomplish this by invoking the preprocessor, the parser, the code generator, and the assembler in separate steps. Example 2–2 shows how you can use a shell program for compiling and assembling a file in one step.

Example 2–1. Method 1 – Invoking Each Tool Individually

1) Invoke the preprocessor; use `program.c` for input:

```
dspcpp program ↵
C Pre-Processor,             Version 5.10
(c) Copyright 1987, 1989 Texas Instruments Incorporated
```

This creates an output file called `program.cpp`.

2) Invoke the parser; use `program.cpp` for input:

```
dspcc program ↵
DSP C Compiler,              Version 5.10
(c) Copyright 1987, 1989 Texas Instruments Incorporated
    "program.c"  ==> abs
```

This creates an output file called `program.if`.

3) Invoke the code generator; use program.if for input:

```
dspcg program ↵
DSP C Codegen,               Version 5.10
(c) Copyright 1987, 1989 Texas Instruments Incorporated
    "program.c"  ==> abs
```

This creates an output file called `program.asm`

4) Assemble `program.asm`

```
dspa program ↵
DSP COFF Assembler,          Version 5.10
(c) Copyright 1987, 1989 Texas Instruments Incorporated
PASS 1
PASS 2

No Errors, No Warnings
```

This creates an output file named `program.obj`

A shell program is shipped as part of the TMS320C25 C compiler package. The shell program expects C source or assembly source files as input, to produce object files that can be linked. To invoke the shell program, enter:

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

dspcl	names the shell program that invokes the tools. If you do not specify filenames or options, the shell displays a help screen with the syntax of the shell command and its options.
options	are single letters preceded by hyphens. Options *are not* case sensitive. Some options have additional fields which follow the option with no intervening spaces. Valid options include:

–c	no linking (negates –z)
–dNAME	predefine NAME
–g	symbolic debugging
–i ***dir***	#include search path
–k	keep `.asm` file
–n	compile only (no asm)
–q	quiet
–qq	super quiet
–s	C source interlist
–uNAME	undefine NAME
–z	link options follow

Preprocessor (–p) options:

–pc	preprocess only
–pp	no #line directives

Runtime (–m) model options:

–ma	aliased variables
–mr	register use info
–mv	volatile variables

Assembler options (–a options)

–al	assembly listing file
–ap	preprocess first
–as	keep local symbols

	–ax cross-reference file
filenames	names C source or assembly files. An extension of `.c` indicates a source file, an extension of `.asm` indicates an assembly file. If you do not specify an extension, the shell assumes it is a C source file (extension of `.c`).
link options	all options following the –z are passed directly to the linker.
object files	names additional object files used in the link step.

The output files have the same name as the input files but with an extension of `obj`. See Section 2.3 for linker options.

Example 2–2 shows how the shell program compiles and assembles a C source file named `program.c` by invoking `dspcpp`, `dspcc`, `dspcg`, and `dspa`.

Example 2–2. Method 2 – Using the Shell Program

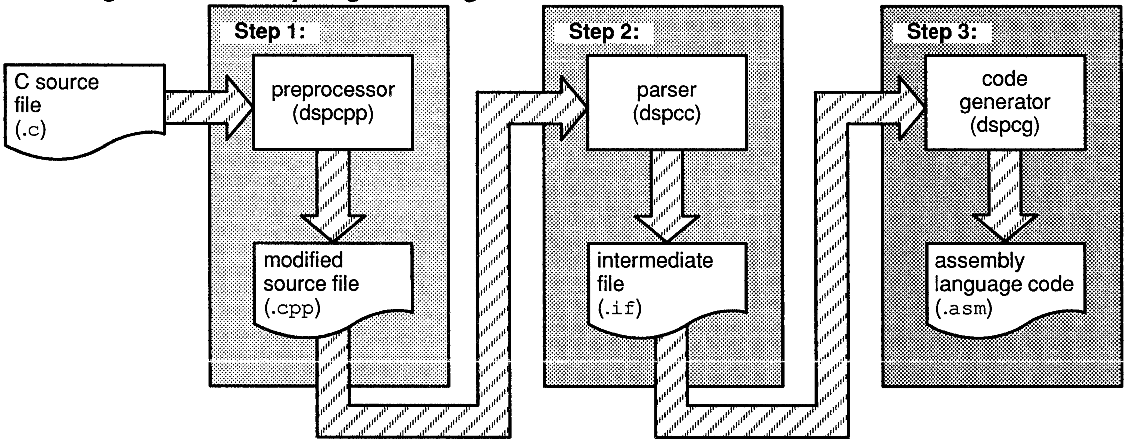
```
dspcl program ↵
[program]
C Pre-Processor,              Version 5.10
(c) Copyright 1987, 1989 Texas Instruments Incorporated
DSP C Compiler,               Version 5.10
(c) Copyright 1987, 1989 Texas Instruments Incorporated
   "program.c"  ==> abs
DSP C Codegen,                Version 5.10
(c) Copyright 1987, 1989 Texas Instruments Incorporated
   "program.c"  ==> abs
DSP COFF Assembler,           Version 5.10
(c) Copyright 1987, 1989 Texas Instruments Incorporated
 PASS 1
 PASS 2

No Errors, No Warnings
```

2.2 Invoking the Compiler Tools Individually

Figure 2–1 illustrates the three-step process of compiling a C program.

Figure 2–1. Compiling a C Program



Step 1: 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.

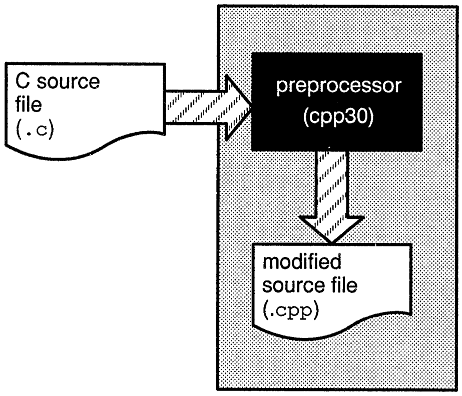
Step 2: The input for the **parser** is the modified source file produced by the preprocessor. The parser produces an intermediate file.

Step 3: The input for the **code generator** is the intermediate file produced by the parser. The code generator produces an assembly language source file.

2.2.1 Preprocessing C Code

The first step in compiling a TMS320C25 C program is invoking the C preprocessor. The preprocessor handles macro definitions and substitutions, #include files, line number directives, and conditional compilation. As Figure 3–2 shows, the preprocessor uses a C source file as input, and produces a modified source file that is used as input for the C parser.

Figure 2–2. Input and Output Files for the C Preprocessor



To invoke the preprocessor, enter:

```
dspcpp  input file [output file] [options]
```

dspcpp 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 prompts 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 are single letters preceded by hyphens. Options *are not* case sensitive. Some options have additional fields that follow the option with no intervening spaces. Valid options include

–c copies comments to the output file. If you don't use this option, the preprocessor strips comments. There is no reason to keep comments unless you plan to inspect the `.cpp` file.

–d*name* [=def]
defines *name* as if it were #defined in a C source file (as in `#define` *name def*). You can use *name* in #if and #ifdef statements without explicitly defining it in the C source. The `=def` is optional; if you don't use it, *name* has a value of 1.

You can use the –d option multiple times to define several names; be sure to separate multiple –d options with spaces.

–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 multiple–i options with spaces. Note that if you don't specify a directory name, the preprocessor ignores the –i option. (For more information about alternate directories, see Section 2.2.1.1.)

–p tells the preprocessor not to produce line number and file information.

–q suppresses the banner and status information.

Note that options can appear anywhere on the command line.

This preprocessor is the same preprocessor that is described in Kernighan and Ritchie; additional information can be found in that book. The preprocessor supports the same preprocessor directives that are described in Kernighan and Ritchie (Appendix B summarizes these directives). All preprocessor directives begin with the **#** character, which must appear in column 1 of the source statement. Any number of blanks or 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 (represented as a C character string).
__TIME__	represents the time that this module was compiled (represented as a C character string).
_dsp	identifies the code as TMS320C25 code; it 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__);
```

could translate to a line such as:

```
printf ("%s %s" , "Jan 14 1988", "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.

2.2.1.1 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 don't provide path information and you enclose *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 set with the environment variable C_DIR.
- ❏ If you don't provide path information and you enclose *filename* in **angle brackets**, the preprocessor searches for the file in
  1) Any directories named with the –i preprocessor option.
  a) Any directories set with the environment variable C_DIR.

Note:

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.

–i Preprocessor Option

The –i preprocessor option names an alternate directory that contains include files. The syntax for the –i option is

dspcpp –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"
```

Note that the include filename is enclosed in double quotes. The complete path/filename for `alt.c` is

- ❑ `c:\dsp\files\alt.c` (DOS systems),
- ❑ `[dsp.files]alt.c` (VMS system),
- ❑ `/dsp/files/alt.c` (UNIX systems), **or**
- ❑ `drive:dsp:files:alt.c` (MPW systems).

This is how you invoke the preprocessor:

DOS:	`dspcpp -ic:\dsp\files source.c`
VMS:	`dspcpp -i[dsp.files] source.c`
UNIX:	`dspcpp -i/dsp/files source.c`
MPW:	`dspcpp -idrive:dsp:files source.c`

The preprocessor first searches for `alt.c` in the current directory, because `source.c` is in the current directory. The preprocessor then searches the directories named with the –i option.

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**	**C_DIR** "*pathname;another pathname ...*"
	export	**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>
```

Assume that the complete path and file information for these files is

- `c:\320\files\alt1.c` and `c:\dsys\alt2.c` (DOS systems)
- `[320.files]alt1.c` and `[dsys]alt2.c` (VMS system),
- `/320/files/alt1.c` and `/dsys/alt2.c` (UNIX systems), **or**
- `drive:320:files:alt1.c` and `drive:dsys:alt2.c` (MPW systems).

This is how you set the environment variable and invoke the preprocessor:

DOS:
```
set C_DIR=c:\dsys; c:\exec\files
dspcpp -ic:\320\files source.c
```

VMS:
```
assign "[dsys]; [exec.files]"  C_DIR
dspcpp -i[320.files] source.c
```

UNIX:
```
setenv  C_DIR "c:/dsys; c:/exec/files"
dspcpp -ic:/320/files source.c
```

MPW:
```
set C_DIR ":dsys;:files"
export C_DIR
dspcpp drive/320/files source.c
```

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

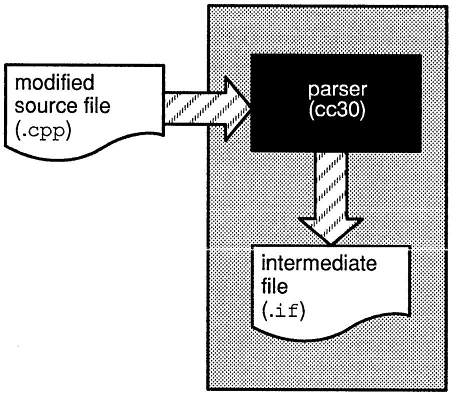
The environment variable remains set until you reboot the system or reset the variable by entering

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

2.2.2 Parsing C Code

The second step in compiling a TMS320C25 C program is invoking 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, if the file contains no preprocessor directives.) Figure 2–3 illustrates this process.

Figure 2–3. Input and Output Files for the C Parser



To invoke the parser, enter

```
dspcc input file [output file] [options]
```

dspcc	is the command that invokes the parser.
input file	names the modified 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 prompts you for one.
output file	names the intermediate file that the parser creates. If you do not supply a filename for the output file, the parser uses the input filename with an extension of `.if`.

options are single letters preceded by hyphens. Options can appear anywhere on the command line and *are not* case sensitive. Valid options include

–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.)
–q	suppresses the banner and status information.

Most errors are fatal; that is, they prevent generation of 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 don't prevent the parser from producing an intermediate file.

As 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 errors. 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.

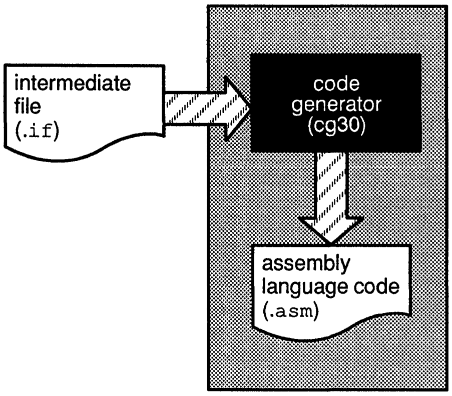
CAUTION

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

2.2.3 Generating Assembly Language Code

The third step in compiling a TMS320C25 C program is invoking the C code generator. As Figure 2–4 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 TMS320C25 assembler. The code generator produces reentrant, relocatable code that can be executed from ROM.

Figure 2–4. Input and Output Files for the C Code Generator



To invoke the code generator, enter:

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

dspcg	is the command that invokes the code generator.
input file	names the intermediate file that the code generator uses as input. The code generator assumes that the input file has an extension of `.if` ; if you supply a different extension, the code generator ignores it. If you don't specify an input file, the code generator prompts you for one.
output file	names the assembly language source file that the code generator creates. If you don't supply a filename for the output file, the code generator uses the input filename with 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 it uses it.
options	are case-sensitive single letters preceded by hyphens. Valid options include:

–o	tells the code generator to place symbolic debugging directives in the output file. See Appendix B of the *TMS320C1x/TMS320C2x Assembly Language Tools User's Guide* for more information about these directives.

–q	suppresses the banner and status information.
–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 without them. Note that if you don't specify the –z option, the code generator deletes the input (intermediate) file.
–r	writes register information to output file in the form of comments.

2.3 Linking a C Program

The TMS320C25 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

> **dsplnk –c** *filenames* [**–o** *output filename*] **–l rts.lib**
> *or*
> **dsplnk –cr** *filenames* [**–o** *output filename*] **–l rts.lib**

dsplnk	is the command that invokes the linker.
–c and **–cr**	are options that tell the linker to use special conventions that are defined by the C environment.
filenames	are object files created by compiling and assembling C programs.
options	Valid options include
	–o allows you to specify the *output filename* including the extension. If this option is not used, the default *output filename* is `a.out`.
	–l allows you to specify an archive library as linker input.
rts.lib	is an archive library that contains C runtime-support functions; it is shipped with the C compiler. Note that you can also link in `flib.lib` (the floating-point support library) or your own object libraries. 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.obj`, `prog2.obj`, and `prog3.obj` (the output file is named `prog.out`):

```
dsplnk -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 directives to customize the allocation process. For more information about the linker, see the *TMS320C1x/TMS320C2x Assembly Language Tools User's Guide*.

2.3.1 Runtime Initialization and Runtime Support

All C programs must be linked with an object module called `boot.obj`. 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:

```
dsplnk -c -l rts.lib ...
```

When a C program begins running, it must execute `boot.obj` first. The symbol `_c_int0` marks the starting point in `boot.obj`; if you use the –c or –cr option, then `_c_int0` is automatically defined as the entry point for the program. If your program begins running from reset, you can set up the reset vector to generate a branch to `_c_int0` so that the TMS320C25 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).
- Calls `main` (the C program).
- Calls `exit` if `main` returns.

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

Note:

You must specify `rts.lib` on the command line.

Chapter 5 describes additional runtime-support functions that are included in `rts.lib`.

2.3.2 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 flib.lib       /* Floating-point library       */
-l matrix.lib     /* Object library               */
```

- The command file in Figure 2–5 first lists several linker options:

  –c is one of the options that can be used to link C code; it tells the linker to use the ROM autoinitialization method.

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

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

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

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

- Finally, the command file lists all the object libraries that the linker must search. (The libraries are specified with the –l linker option.) Because this is a C program, the runtime-support library `rts.lib` **must** be included. If a program uses floating-point routines, it must also link in `flib.lib` (the floating-point library). 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.

To link the program, enter

```
dsplnk link.cmd
```

This example uses the default memory allocation described in Section 8.8 of the *TMS320C1x/TMS320C2x Assembly Language Tools User's Guide.* If you want to specify different MEMORY and SECTIONS definitions, refer to that user's guide.

2.3.3 Autoinitialization (ROM and RAM Models)

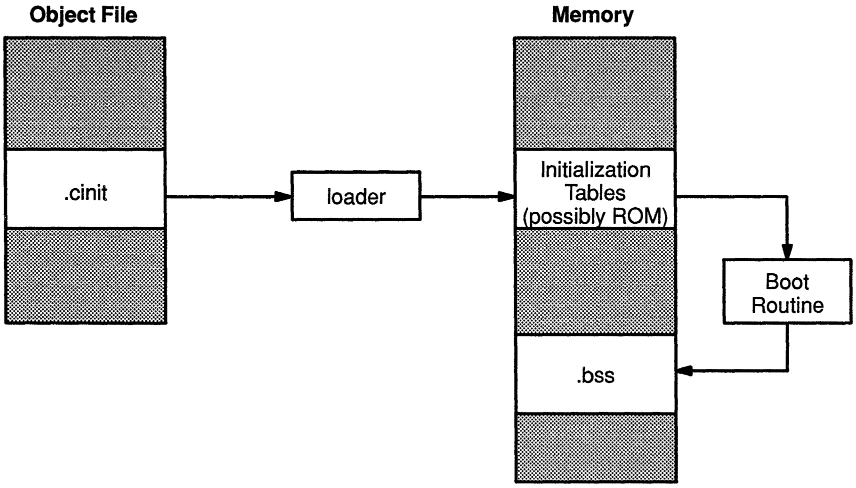
The C compiler produces tables of data for autoinitializing global variables. (Section 4.8.2, page 4–19, discusses the format of these tables.) These tables are in a named section called **.cinit**. The initialization tables can be used in either of two ways:

- **ROM Model** (–c option)

  Global variables are initialized at *runtime*. 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.

Figure 2–6 illustrates the ROM autoinitialization model.

Figure 2–6. ROM Model of Autoinitialization



❑ **RAM Model** (–cr option)

Global variables are initialized at *load time.* This enhances performance by reducing boot time and saving memory used by the initialization tables. (Note that you must use a smart loader to take advantage of the RAM model of autoinitialization.)

When you use –cr, the linker marks the .cinit section with a special attribute. This attribute tells the linker *not* to allocate the .cinit section into memory. The linker also sets the `cinit` symbol to -1; this tells the C boot routine that initialization tables *are not* present in memory. Thus, no run-time initialization is performed at boot time.

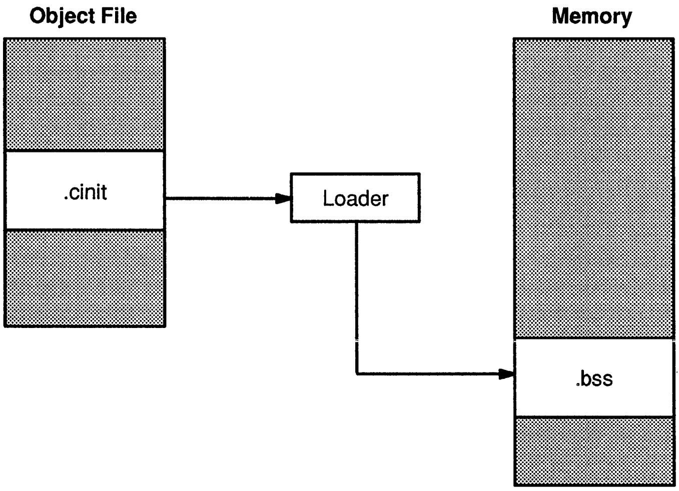
When the program is loaded, the loader must be able to

- Detect the presence of the .cinit section in the object file.
- Detect the presence of the attribute that tells it not to load the .cinit section.
- Understand the format of the initialization tables (described in Section 4.8.2 on page 4–19).

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

Figure 2–7 illustrates the RAM autoinitialization model.

Figure 2–7. RAM Model of Autoinitialization



2.3.4 The –c and –cr Linker Options

The following list outlines what happens when you invoke the linker with the –c or –cr option. These are the linking conventions required for C programs.

- ❑ The symbol _c_int0 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_int0 is automatically referenced; this ensures that `boot.obj` is automatically linked in from the runtime-support library.
- ❑ The .cinit output section is padded with a termination record so that the boot routine (ROM model) or the loader (RAM model) knows when to stop reading the initialization tables.
- ❑ In the ROM model (–c option), the linker defines the global symbol `cinit` as the starting address of the .cinit section. The C boot routine uses this symbol as the starting point for autoinitialization.
- ❑ In the RAM model (–cr option):
  - ■ The linker sets the symbol `cinit` to –1. This indicates that the initialization tables are not in memory, so no initialization is performed at runtime.
  - ■ The STYP_COPY flag (010h) is set in the .cinit section header. STYP_COPY is the special attribute that tells the loader to perform

autoinitialization directly and not to load the .cinit section into memory. The linker does not allocate space in memory for the .cinit section.

2.4 Archiving a C Program

An archive file (or library) is a partitioned file that contains complete files as members. The TMS320C1x/TMS320C2x archiver is a software utility that allows you to collect a group of files together 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 *TMS320C1x/TMS320C2x 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 file management and portability. The library `rts.src` is an example of an archive file that contains source files.

For more information about the archiver, see the *TMS320C1x/TMS320C2x Assembly Language Tools User's Guide*.

Chapter 3

TMS320C25 C Language

The C language that the TMS320C25 C compiler supports is based on the UNIX System V C language that is described by Kernighan and Ritchie, with several additions and enhancements. The most significant differences are

- ❑ The addition of data type *enum*.
- ❑ Unique member names are not required in structures.
- ❑ Structures and unions can be passed as parameters to functions and assigned directly.

This chapter compares the C language compiled by TMS320C25 C to the C language described by Kernighan and Ritchie. It presents only the *differences* in the two forms of the C language. The TMS320C25 C compiler supports standard Kernighan and Ritchie C except as noted in this section.

Throughout this chapter, references to Kernighan and Ritchie's *C Reference Manual* (Appendix A of *The C Programming Language*) are shown in the left margin.

Topics in this chapter include

Section **Page**

3.1 Identifiers, Keywords, and Constants 3-2
3.2 Data Types 3-4
3.3 Object Alignment 3-6
3.4 Expressions 3-6
3.5 Declarations 3-6
3.6 Initialization of Static and Global Variables 3-9
3.7 asm Statement (Inline Assembly Language) 3-9

3.1 Identifiers, Keywords, and Constants

K&R 2.2 – Identifiers

- In TMS320C25 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 for identifier names in all TMS320C25 C tools. This also applies to external names.

K&R 2.3 – Keywords

TMS320C25 C reserves **three additional keywords**:

asm
void
enum

K&R 2.4.1 – Integer Constants

- All integer constants are of type *int* (signed, 16 bits long) unless they have an *L* or *U* suffix. If the compiler encounters an invalid digit in a constant (such as 8 or 9 in an octal constant), it issues a warning message.
- You can append a letter suffix to an integer constant to specify what kind of integer it is:
  - Use *U* as a suffix to declare an **unsigned integer constant**.
  - Use *L* as a suffix to declare a **long constant**.
  - Combine the suffixes to declare an **unsigned long integer constant**.

  Note that the 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 2.4.3 – Character Constants

In addition to the escape codes listed in K&R, the TMS320C25 C compiler **recognizes the escape code \v** in character and string constants as a vertical tab character (ASCII code 11).

K&R 2.4.4 – Floating-Point Constants

In TMS320C25 C, floats, doubles, and long doubles are single-precision (32-bit) floating-point numbers. To provide compatibility with ANSI standard C, the compiler allows you to append one of two suffixes to a floating-point constant:

- *L* identifies long double constants in ANSI C.
- *F* identifies floats in ANSI C.

You can use either upper or lower case letters. Examples of floating-point constants include:

```
1.234;          /* double        */
1.0e14f;        /* float         */
3.14159L;       /* double        */
```

Note that using a suffix does not change the way the compiler treats the number.

Added Type – Enumeration Constants

Enumeration constants are **an additional type of integer constant** that is not described by K&R. An identifier declared as an enumerator can be used in the same manner that an integer constant can be used. (For more information about enumerators, see Section 3.5 on page 3–6.)

K&R 2.5 – String Constants

- K&R C does not limit string constant length; *however*, TMS320C25 C **limits the length of string constants to 255 bytes**.
- All characters after an embedded null byte in a string constant are ignored; in other words, the first null byte terminates the string. However, 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. However, this does not apply to strings used for autoinitialization of arrays of characters.

3.2 Data Types

K&R 4.0 – Equivalent Types

- The *char* data type is signed. A separate type, *unsigned char,* is also supported.
- *long, int, short,* and *char.* are all functionally equivalent types. Any of these types can be declared unsigned.
- *float* and *double* are functionally equivalent types.
- 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 a 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             */
p = &i;                 /* Legal             */
c = malloc();           /* Legal             */
i = *p;                 /* Illegal           */
i = *(int *)p;          /* Legal             */
```

K&R 4.0 – Derived Types

TMS320C25 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, the maximum allowed.

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

Also, 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[][];
```

K&R 2.6 – Summary of TMS320C25 Data Types

Type	Size
char	16 bits, signed
unsigned char	16 bits
short	16 bits
unsigned short	16 bits
int	16 bits
unsigned int	16 bits
long	16 bits
unsigned long	16 bits
pointers	16 bits
float double	32 bits Range: $\pm 1.17549435 \times 10^{(-38)}$ through $\pm 3.40282347 \times 10^{38}$
enum	16 bits

3.3 Object Alignment

All objects except bit fields are aligned on 16-bit (one word) boundaries. Bit fields are always unsigned and can be from 1 to 16 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 the 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 16 (16 is the number of bits in a character). The *sizeof* operator can be legally applied to enum objects and bit fields; if the result is not an integer, it is rounded up to the nearest integer.

3.5 Declarations

K&R 8.1 – Register Storage Class Specifiers

- The first two local objects declared as type *register* in a function will be allocated to TMS320C25 registers.
- Any type variable (int, unsigned, char, pointer, float, array, etc.) can be declared as a register, since this stores the *address* of that variable in the register. Formal parameters to functions can also be declared as type register.
- Register declarations are ignored in nested blocks.

For more information about register variables, see Section 4.2.3 on page 4–7.

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.
- TMS320C25 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, K&R 10 – Type Specifiers in Function Declarations

- Structures and unions can be used as parameters to functions and can be directly assigned.
- Formal parameters to a function can be declared as type *struct* or *enum* (in addition to the normal function declarations), since TMS320C25 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 16 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 TMS320C25 C, **members of different structures or unions can have the same name.** However, this requires that references to any member of a structure or union be fully qualified through all levels of nesting.
- TMS320C25 C allows assignment to and from structures and passing structures as parameters.
- K&R contains a statement about the compiler determining the type of a structure reference by the member name. Because TMS320C25 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 TMS320C25 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. You can use this tag 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 enumerator 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)
```

Sixteen 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 TMS320C25 C is that **in TMS320C25 C, external and static variables are not preinitialized to zero** unless the program explicitly does so or it is specified by the linker.

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

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

3.7 asm Statement (Inline Assembly Language)

Additional Statement

TMS320C25 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.

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 statement 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.

In addition, do not use the asm statement to insert assembler directives which would change the assembly environment.

3.8 Lexical Scope Rules

K&R 11.1

The lexical scope rules stated in K&R apply to TMS320C25 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.

Chapter 4

Runtime Environment

This section describes the TMS320C25 C runtime environment. To ensure successful execution of C programs, it is critical that all runtime code maintain this environment. If you write assembly language functions that interface to C code, follow the guidelines in this chapter.

Topics in this chapter include:

Section **Page**
4.1 Memory Model 4-2
4.2 Register Conventions 4-6
4.3 Function Call Conventions 4-8
4.4 Interfacing C with Assembly Language 4-11
4.5 Interrupt Handling 4-16
4.6 Integer Expression Analysis 4-18
4.7 Floating-Point Expression Analysis 4-19
4.8 System Initialization 4-20

4.1 Memory Model

The TMS320C25 treats memory as two linear blocks of program memory and data memory:

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

Each block of code or data generated by a C program is placed into a contiguous block in the appropriate memory space.

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 available, about any locations not available for code or data (holes), or about any locations reserved for I/O or control purposes. The compiler produces relocatable code that allows the linker to allocate code and data into the appropriate memory spaces. For example, you can use the linker to allocate global variables into fast internal RAM, or allocate executable code into internal ROM. Each block of code or data could be allocated individually into memory, but this is not a general practice (an exception to this is memory-mapped I/O, although physical memory locations can be accessed with C pointer types).

4.1.1 Sections

The compiler produces three relocatable blocks of code and data; these blocks, called **sections**, can be allocated into memory in a variety of ways, to conform to a variety of system configurations. For more information about sections, please read Section 3 (the Introduction to Common Object File Format) of the *TMS320C1x/TMS320C2x Assembly Language Tools User's Guide*.

There are two basic types of sections:

- **Initialized sections** contain data or executable code.
- **Uninitialized sections** reserve space in memory (usually RAM). A program can use this space at run time for creating and storing variables.

The C compiler creates two initialized sections, .text and .cinit; it creates one uninitialized section, .bss.

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

- The **.bss section** is an uninitialized section; in a C program, it serves three purposes:
  - It reserves space for global and static variables. At boot time, the the C boot routine copies data out of the .cinit section (which may be in ROM) and uses it for initializing variables in .bss.
  - It reserves space for the system stack, which is used to pass arguments to functions and to allocate local variables.
  - It reserves space for use by the dynamic memory functions (malloc, calloc, and realloc).

Note that the *assembler* creates three default sections (.text, .bss, and .data); the C compiler, however, does not create a .data section because the C environment does not use this section.

The linker takes the individual sections from different modules and combines sections with the same name to create three output sections. The complete program is made up of these three output sections, plus the .data section that the assembler creates. 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 section should be linked into some type of RAM.

For more information about allocating sections into memory, see Chapter 9 (the Linker Description) of the *TMS320C1x/TMS320C2x Assembly Language Tools User's Guide.*

4.1.2 Allocating Memory for Static and Global Variables

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

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

4.1.3 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 from ROM to the initialized variables in .bss (RAM).

In situations where a program is loaded directly from an object file into memory and then run, you can avoid having the .cinit section occupy space in memory. A loader can read the initialization tables directly from the object file (instead of from ROM) and perform the initialization directly at load time (instead of at run time). You can specify this *to the linker* by using the –cr linker option. For more information, see Section 2.3.3 on page 2-17 .

4.1.4 Managing the Runtime Stack

The C compiler uses a stack to:

- ❑ Allocate local variables.
- ❑ Pass arguments to functions, **and**
- ❑ Save the processor status.

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

AR1 is the **stack pointer** (**SP**); it points to the current top of the stack *or* to the word that follows the current top of the stack.

AR0 is the **frame pointer** (**FP**); it points to the beginning of the current frame. Each function invocation creates a new frame at the top of the stack, from which local and temporary variables are allocated.

The C environment manipulates these registers automatically; if you write any assembly language routines that use the runtime stack, be sure to use these registers correctly. (For more information about using these registers, see Section 4.2. on page 4–6; for more information about the stack, see Section 4.3. on page 4–8.)

The C initialization routine, `boot.asm`, allocates memory for the stack in .bss. The routine also defines a constant named STACK_SIZE that determines the amount of space reserved for the stack. The default stack size is 1000 words. You can change the stack size 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`.

Note that the internal TMS320C25 stack is used only temporarily. Return addresses that are pushed onto it by CALL instructions are popped off and pushed onto the runtime stack.

4.1.5 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.

A C module called `memory.c` reserves space for this memory pool in the .bss section. The module also defines a constant named `MEMORY_SIZE` that determines the size of the memory pool; the default size is 1000 words. You can change the size of the memory pool by following these steps:

1) Extract `memory.c` from the source library `rts.src`.
2) Edit `memory.c`; change the value of the constant `MEMORY_SIZE` to the desired memory pool size.
3) Recompile and assemble `memory.c` and replace the resulting object file, `memory.obj`, in the object library `rts.lib`.

4.1.6 Packing Structures and Manipulating Fields

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

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

4.2 Register Conventions

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

4.2.1 Dedicated Registers

The C environment reserves three registers. **Do not** modify these registers in any other manner than that described in Section 4.3, Function Call Conventions (page 4–8).

AR0 is the **frame pointer**. It points to the current activation record (the beginning of the current frame).

AR1 is the **stack pointer**. It points to the top of the runtime stack *or* to the word that follows the top of the stack.

AR2 is the **local variable pointer**. It is used for calculating the address of local variables.

4.2.2 Using Registers

The auxiliary registers, the accumulator, the T and P registers, and miscellaneous status registers can be used by assembly language functions; be sure to follow these rules:

- **Auxiliary Registers (ARP and AR0–AR7)**
  - The ARP **must** contain a 1 when a function is entered and when a function returns. It may contain other values during function execution.
  - Registers AR0 and AR1 may be modified during function execution, but they must be restored.
  - Registers AR2, AR3, AR4, and AR5 may be modified and do not need to be restored.
  - Registers AR6 and AR7 are used for register variables. If they are modified, they must be saved and restored.
- **Status Register**
  - The C compiler assumes that the PM status bits *are always set to 0.* If you change these bits, you must set them to 0 before returning from the function.
  - You can change the following status fields without restoring them: DP, C, FSM, HM, INTM, OV, OVM, SXM, TC, TXM, ARB, CNV, FO, and XF.

- **Other Registers**
  - You can use the accumulator without saving and restoring the original value. It is also used to return integers, pointers, and floating-point values.
  - You can use the P and T registers without saving and restoring their original values.

4.2.3 Register Variables

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

The compiler uses AR6 and AR7 for register variables:

- AR6 is assigned to the first register variable.
- AR7 is assigned to the second variable.

Note that the **address** of the variable is placed into the allocated register to simplify access. Thus, *any* type of variable may be used as a register variable.

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

4.3 Function Call Conventions

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

4.3.1 Passing Parameters to a Function

If you pass parameters to a C function, you **must** follow these rules:

- Push them on the runtime stack before you call the function.
- Pass arguments in reverse order; that is, push the first (leftmost) parameter last, and push the last (rightmost) parameter first. This allows you to pass a variable number of parameters.
- If an argument is a floating-point value, push the least significant word first.
- Pass structures as multiples of words (they can be lengthened if necessary).

4.3.2 Local Frame Generation

When a function is called, the compiler builds a frame (or activation record) to store information on the runtime stack. The current function's frame is called the local frame. The C environment uses the local frame for saving information about the caller, passing arguments, and generating local variables. Each time you call a function, a new frame is generated to store information about that function. When you return from a called function, the caller's frame is still on the stack so that you can continue to use that information.

Register AR1 is the SP (stack pointer) and register AR0 is the frame pointer (FP). The SP points to the next available word on the stack; the FP points to the local frame.

The compiler performs the following tasks when it builds the local frame:

1) Pops the return address (old PC value) from the TMS320C25 internal stack and pushes it on the runtime stack.
2) Pushes the contents of the old FP onto the runtime stack and sets the new FP to the current SP.
3) Increments the SP by the number of words needed to hold local variables, plus one word at the beginning of the frame for temporary storage. (In the following code segment, the symbol `SIZE` represents the amount of this space.)

4) If the function uses AR6 and AR7 (as register variables), it pushes their contents onto the stack and then loads them with the addresses of appropriate local variables.

Here is the TMS320C25 code that performs these tasks:

```
POPD *+              ; Pop return address
SAR  AR0,  *+        ; Push on system stack
SAR  AR1,   *        ; Save old FP
LARK AR0,  SIZE
LAR  AR0,  *0+       ; FP = old SP, SP += SIZE
SAR  AR6,  *+        ; Save AR6
SAR  AR7,  *+        ; Save AR7
```

There are several points to keep in mind about this method of frame generation:

- When a function is entered, the compiler assumes that the ARP points to the stack pointer (AR1).
- There is no separate pointer to the argument list; the frame pointer is used with negative offsets to point to the arguments, and with positive offsets to point to local variables.
- The frame pointer (AR0) points to a single word, which is allocated before the local variables. This word is used as a temporary memory location to allow register-to-register transfers and is **essential** to creating reentrant C functions. Note that this memory location can **always** be directly accessed through AR0.
- The compiler uses AR2 to calculate the address of local variables. Generally, the offset of the local variables is placed in AR2, and AR0 is added to it. This value is not preserved across function calls.

4.3.3 Function Termination

When a function is terminated, it must perform the following tasks to restore the calling environment:

1) Handle return values that will be passed to the caller.
2) Restore AR6 and AR7 if it used them.
3) Deallocate space used for local variables and temporary memory. (In the following code segment, the symbol `SIZE` represents the amount of this space.)
4) Restore the old frame pointer.
5) Push the return address onto the TMS320C25 stack and return to the caller.

Here is the TMS320C25 code that performs these tasks:

```
LAR      AR7, *-     ; Restore AR7
LAR      AR6, *-     ; Restore AR6
SBRK     SIZE        : Deallocate frame
LAR      AR0, *-     ; Restore FP
PSHD     *           ; Put return address on internal stack
RET                  ; Return to caller
```

Note that

- All return values are returned in the accumulator. Integers and pointers are returned (properly sign extended) in the 16 LSBs of the accumulator. Floating-point values use all 32 bits of the accumulator.
- Because the accumulator contains the return value, it must not be modified by epilog code.
- Arguments are not popped off the stack by the called function; they must be popped by the caller. This allows you to pass any number of arguments to a function, and the function need not know how many were passed.
- Upon return from a function, the ARP points to AR1.
- Structures cannot be returned from functions.

4.4 Interfacing C with Assembly Language

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

- ❏ Use separate modules of assembled code and link them with compiled C modules (see Section 4.4.1). This is the most versatile method.
- ❏ Use inline assembly language, embedded directly in the C source (see Section 4.4.2, page 4–14).
- ❏ Modify the assembly language code that the compiler produces (see Section 4.4.3, page 4–15).

4.4.1 Assembly Language Modules

Interfacing with assembly language functions is straightforward if you follow the calling conventions defined in Section 4.3 and the register conventions defined in Section 4.2. C code can access variables and call functions defined in assembly language, and assembly code can access C variables and call C functions.

Follow these guidelines to interface assembly language and C:

1) All functions, whether they are written in C or assembly language, must follow the conventions outlined in Section 4.2. (page 4–6).
2) You must preserve any dedicated registers modified by a function; dedicated registers include

AR0 (FP)	AR6
AR1 (SP)	AR7

   Note that if 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). Any register that is not dedicated can be used freely without being saved. Upon return from the function, the ARP must be 1.
3) Interrupt routines must save **all** the registers they use. (For more information about interrupt handling, see Section 4.5, page 4–16.)
4) When calling a C function from assembly language, push any arguments onto the stack in reverse order. Pop them off after calling the function.
5) When calling C functions, remember that only the dedicated registers listed above are preserved. C functions can change the contents of any other register.
6) Functions must return values correctly according to their C declarations. Integers and pointers are returned in the accumulator, and floating-point values are returned on the stack.

7) No assembly language module should use the .cinit section for any purpose other than autoinitialization of global variables. The C startup routine in `boot.asm` assumes that the .cinit section consists **entirely** of initialization tables.

CAUTION

Disrupting the tables by putting other information in .cinit can cause unpredictable results.

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

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

   Similarly, to access a C function or object from assembly, declare the C object with .global, thus creating an undefined external reference that the linker will resolve.

4.4.1.1 An Example of an Assembly Language Function

The example in Figure 4–1 illustrates a C function called `main`, which calls an assembly language function called `asmfunc`. The `asmfunc` function takes its single argument, adds it to the C global variable called `gvar`, and returns the result.

Figure 4–1. An Assembly Language Function

(a) C program

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

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

Figure 4–1. An Assembly Language Function (Continued)

(b) Assembly language program

```
_asmfunc:
    POPD     *+            ; Move return address to C stack
    SAR  AR0, *+           ; Save FP
    SAR  AR1, *            ; Save SP
    LARK AR0, 1            ; Size of frame
    LAR  AR0, *0+, AR2     ; Set up FP and SP

    LDPK _gvar             ; Point to gvar
    SSXM                   ; Set sign extension
    LAC  _gvar             ; Load gvar
    LARK AR2, -3           ; Offset of argument
    MAR  *0+               ; Point to argument
    ADD  *, AR0            ; Add arg to gvar
    SACL _gvar             ; Save in gvar

    LARP AR1               ; Pop off frame
    SBRK 2
    LAR  AR0, *-           ; Restore frame pointer
    PSHD *                 ; Move return addr to C25 stack
    RET
```

In the C program in Figure 4–1, the *extern* declaration of `asmfunc` is optional, since the function returns an int. Like C functions, assembly functions need be declared only if they return noninteger values. In the assembly language code in Figure 4–1, note the underscores on all the C symbol names used in the assembly code.

4.4.1.2 Defining Variables in Assembly Language

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

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

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

Figure 4–2. Accessing from C a Variable Defined in .bss

(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
```

Figure 4–2. Accessing from C a Variable Defined in .bss (Continued)

(b) C program

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

You may not always want a variable to be in the .bss section; for example, a common situation is a lookup table defined in assembly that you don't want to put in RAM. In this case, you must define a pointer to the object and access it indirectly from C.

The first step is to define the object; it is helpful (but not necessary) to put it in its own initialized section. Declare a global label that points to the beginning of the object, and then the object can be linked anywhere into the memory space. To access it in C, you must declare 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 4–3 shows an example that accesses a variable that is not defined in .bss.

Figure 4–3. Accessing from C a Variable not Defined in .bss

(a) Assembly language program

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

(b) C program

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

4.4.2 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.

CAUTION

When you use asm statements, be extremely careful not to disrupt the C environment. The compiler does not check or analyze the inserted instructions.

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

4.4.3 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 warnings in Section 4.4.2 about disrupting the C environment also apply to modification of compiler output.

4.5 Interrupt Handling

Interrupts can be handled *directly* with C functions by using the following reserved function names, which are associated with the indicated interrupt:

c_int0	system reset interrupt
c_int1	external interrupt 0
c_int2	external interrupt 1
c_int3	external interrupt 2
c_int4	internal timer interrupt
c_int5	serial port receive interrupt
c_int6	serial port send interrupt

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

When an interrupt handling function is entered, the runtime-support function I$$SAVE is called to save the complete context of the interrupted function. **All** registers are saved. Upon return from the interrupt handling function, the runtime-support function I$$REST is called to restore the environment and return to the interrupted function.

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

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

- ❑ It is your responsibility to handle any special masking of interrupts (via the IMR register). You can use inline assembly to enable or disable the interrupts and modify the IMR register without corrupting the C environment.
- ❑ An interrupt handling routine cannot have arguments. If any are declared, they are ignored.
- ❑ An interrupt handling routine can be called by normal C code, but it is inefficient to do this because all the registers are saved.
- ❑ An interrupt handling routine can handle a single interrupt or multiple interrupts. The compiler does not generate code that is specific to a certain interrupt, except for c_int0, which is the system reset interrupt. When you enter this routine, you cannot assume that the runtime stack is set up; therefore, you *cannot allocate local variables*, and you *cannot save any information on the runtime stack.*

- ❑ To associate an interrupt routine with an interrupt, a branch must be placed in the appropriate interrupt vector. The assembler and linker can easily accomplish this by creating a simple table of branch instructions.
- ❑ In assembly language, remember to precede the symbol name with an underscore. For example, refer to `c_int0` as `_c_int0`.

4.6 Integer Expression Analysis

Most integer expressions are analyzed in the 16 LSBs of the accumulator or in the P or T registers. TMS320C25 C follows Kernighan and Ritchie standard precedence rules for evaluating expressions; however, the order of expression analysis is based on the complexity of the operands. (Note that this does not apply to operators such as the comma, &&, or ||, which specify a particular order of analysis.) The following rules determine which portion of an expression is evaluated first:

1) If the expression contains a division or modulus operation, the *right-hand side of the expression* is evaluated first. **otherwise,**
2) If either operand contains a function call, the *function call* is evaluated first. **otherwise,**
3) If either operand contains a multiplication expression, the *multiplication* is evaluated first. **otherwise,**
4) If rules 1—3 do not apply, *the operand that appears to be more complex* is evaluated first. (Complexity is based on the number and types of operations.)

4.6.1 Arithmetic Overflow and Underflow

The TMS320C25 performs 32-bit math with 16-bit values; thus, **arithmetic overflow and underflow cannot be handled in a predictable manner.** If code depends on a particular type of overflow/underflow handling, there is no guarantee that this code will execute correctly. Generally, it is a good practice to avoid such code because it is not portable.

4.6.2 Integer Division and Modulus

The TMS320C25 does not directly support integer division, so all division and modulus operations are performed through calls to runtime-support routines. These functions push the right-hand portion (divisor) of the operation onto the stack, and then place the left-hand portion (dividend) into the 16 LSBs of the accumulator. The function places the result in the accumulator.

4.7 Floating-Point Expression Analysis

The TMS320C25 C compiler represents floating-point values as IEEE single-precision numbers. Both single-precision and double-precision floating-point numbers are represented as 32-bit values; there is no difference between the two formats.

The TMS320C25 floating-point library, `flib.lib`, contains a custom-coded set of floating-point math functions that support

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

These functions *do not* follow standard C calling conventions. Instead, the compiler pushes the arguments onto the runtime stack and generates a call to a floating-point function. The function pops the arguments, performs the operation, and pushes the result onto the stack.

Some floating-point functions expect integer arguments or return integer values. For floating-point functions, all integers are passed and returned in the 16 LSBs of the accumulator.

The compiler does not recognize the internal format of floating-point numbers; the only restriction is the data size of the number. This allows you to customize a floating-point package for a particular environment.

4.8 System Initialization

Before you can run a C program, the C runtime environment must be created. This task is performed by the C boot routine, which is a function called c_int0. The runtime-support source library contains the source to this routine in a module called `boot.asm`.

The c_int0 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 linker option and include `rts.lib` as a linker input file. When you use –c or –cr, the linker sets the entry point in the executable output module to the symbol _c_int0. The C boot routine performs three tasks:

- It reserves space in .bss for the runtime stack and sets up the initial stack pointer and frame pointer.
- It autoinitializes global variables by copying the data from the initialization tables in .cinit to the storage allocated for the variables in .bss.

  Note that in the RAM autoinitialization model, a loader performs this step before the program runs (it is not performed by the boot routine).
- It calls the function `main` to begin running the C program.

You can replace or modify the C boot routine, if necessary, to meet the particular requirements of your system. However, you must be sure to perform the preceding three steps correctly to initialize the C environment properly.

4.8.1 Runtime Stack

The runtime stack is allocated in a single contiguous block of memory and grows up from low addresses to higher addresses. Register AR1 usually points to the next available word in the stack (top of the stack plus one word). The compiler can use this word as a temporary memory location, so it must be saved by interrupt routines.

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

The stack is allocated by the runtime-support module `boot.asm` as a static array named SYS_STACK. (`boot.asm` is a member of the `rts.src` library.) Set the size of this array to the size required for the stack. Other methods of stack allocation can be used, but you must then rewrite the c_int0 boot function to comprehend the new configuration.

4.8.2 Autoinitialization

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

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

Offset	Contents
Word 0	Length of initialization data (in words)
Word 1	Address of variable to be initialized
Word 2...*n*	Data to be copied into the variable

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

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

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

The initialization tables would appear as follows:

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

The .cinit section contains **only** initialization tables in this format. When interfacing assembly language modules, do not use the .cinit section for any other use.

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

4.8.2.1 ROM Initialization Model

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

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

4.8.2.2 RAM Initialization Model

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

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

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

Chapter 5

Runtime-Support Functions

Some of the tasks that a C program must perform (such as memory allocation, string conversion, and string searches) are not part of the C language. The runtime-support functions, which are included with the C compiler, are standard 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 an input file 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.

You can find these topics on the following pages:

Section **Page**
5.1 Header Files 5-2
5.2 Summary of Runtime-Support Functions and Macros 5-8
5.3 Functions Reference 5-14

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.

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

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

To use a runtime-support function, you must first use the #include preprocessor directive to include the header file that declares the function. For example, the isdigit function is declared by the `ctype.h` header. Before you can use the isdigit function, you must first include 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 5.1.8 describe the header files included with the TMS320C25 C compiler. Section 5.2, page 5-8, lists the functions that these headers declare.

5.1.1 Diagnostic Messages (`assert.h`)

The `assert.h` header defines the assert macro, which provides a standard method for allowing programs to create diagnostic failure messages at runtime. The assert macro tests a runtime expression.

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

The `assert.h` header refers to another macro named NDEBUG (`assert.h` does not define NDEBUG – you must define it). If NDEBUG *is* defined as a macro name when you include `assert.h`, then the assert macro is turned off and has no effect.

5.1.2 Character Typing and Conversion (`ctype.h`)

The `ctype.h` header declares functions and defines macros that test or convert ASCII characters:

❏ The character-testing functions have names with the format is*xxxx* (for example, isdigit). These functions return *true* (1) or *false* (0).

❏ The character-conversion functions have names in the format toxxxx (for example, toupper). They convert a character to lowercase, uppercase, or ASCII, and return the converted character.

The `ctype.h` header also defines macro definitions that perform these same operations. The typing functions expand to a lookup operation in an array of flags (this array is defined in `ctype.h`). The macros have the same names as the corresponding functions, prefixed with an underscore (for example, *_isascii*).

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 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 Limits for Characters and Integers

Macro	Value	Description
CHAR_BIT	16	Maximum number of bits for the smallest object that is not a bit field
SCHAR_MIN	–32768	Minimum value for a signed char
SCHAR_MAX	32767	Maximum value for a signed char
UCHAR_MAX	65535	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	–32768	Minimum value for a short int
SHRT_MAX	32767	Maximum value for a short int
USHRT_MAX	65535	Maximum value for an unsigned short int
INT_MIN	–32768	Minimum value for an int
INT_MAX	32767	Maximum value for an int
UINT_MAX	65535	Maximum value for an unsigned int
LONG_MIN	–32767	Minimum value for a long int
LONG_MAX	32767	Maximum value for a long int
ULONG_MAX	65535	Maximum value for an unsigned long int

Table 5–2. Macros That Supply Limits for Floating-Point Numbers

Macro	Value	Description
FLT_RADIX	2	Base or radix of exponent representation
FLT_ROUNDS	1	Rounding mode for floating-point addition (rounds to nearest integer)
FLT_DIG DBL_DIG LDBL_DIG	6	Number of decimal digits of precision for a float, double, or long double
FLT_MANT_DIG DBL_MANT_DIG LDBL_DIG	24	Number of base-FLT_RADIX digits in the mantissa of a float, double, or long double
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	–125	Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized float, double, or long double
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	128	Maximum integer such that FLT_RADIX raised to that power minus 1 is a representive finite float, double, or long double
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	1.19209290E–07F	Minimum positive float, double, or long double number *x* such that $1.0 + x \neq 1.0$
FLT_MIN DBL_MIN LDBL_MIN	1.17549435E–38F	Minimum positive float, double, or long double
FLT_MAX DBL_MAX LDBL_MAX	3.40282347E+38F	Maximum float, double, or long double
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	–37	Minimum negative integer such that 10 raised to that power is in the range of normalized floats, doubles, or long doubles
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	38	Maximum integers such that 10 raised to that power is in the range of representable finite floats, doubles, or long doubles

Key to prefixes:

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

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

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

(In TMS320C25 C, a double is equivalent to a float and is actually a single-precision number.)

Most of the math functions expect arguments to be within a certain range. The `math.h` header also defines three macros that can be used with the math functions for reporting range errors:

- *EDOM*
- *ERANGE*
- *HUGE_VAL*

If the value of an argument is outside the expected range, a *domain error* occurs, and the errno macro is set to the value of the EDOM macro. If a result cannot be represented (usually because it's too large), or if a result overflows, a function returns the value of the HUGE_VAL macro, and the errno macro is set to the value of the ERANGE macro.

If the result of a floating-point math function cannot be represented as a floating-point value, a *range error* occurs.

5.1.5 Variable Arguments (stdarg.h)

A function can be called with a variable number of arguments with different types. The `stdarg.h` header declares

- A type, *va_list,* **and**
- Three macros, *va_start, va_arg,* and *va_end.*

A variable-argument function can use the objects declared by `stdarg.h` to advance through a list of arguments when the number and types of arguments that are passed to it may vary.

The *va_list* type is an array type that can hold information for the macros.

5.1.6 Standard Definitions (stddef.h)

The `stddef.h` header defines two types and three macros. The types include

- *ptrdiff_t* is a signed integer that is the result type from subtracting two pointers .
- *size_t* is an unsigned integer that is the result type of the *sizeof* operator.

The macros include

- *NULL,* which expands to a null pointer constant
- *offsetof(type, identifier),* 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*).
- *errno,* which expands to the variable `_errno`. This macro is used to report errors from runtime-support functions.

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

5.1.7 General Utilities (`stdlib.h`)

The `stdlib.h` header declares several macros, as well as two types. The macros include

- ❑ EXIT_FAILURE
- ❑ EXIT_SUCCESS, **and**
- ❑ RAND_MAX.

The `stdlib.h` header declares several kinds of 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 macro memory_size in the runtime-support module `memory.c`. (This module is defined in the file `rts.src`.) By default, the amount of memory available for allocation is 1000 words. You can change this amount by modifying the memory_size macro and recompiling `memory.c`.
- ❑ 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.
- ❑ Function-exit functions that allow you to terminate functions normally or abnormally.

5.1.8 String Functions (`string.h`)

The `string.h` header declares 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.

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

5.2 Summary of Runtime-Support Functions and Macros

Function or Macro	Find on page . . .
Error Message Macro	5-8
Character Typing and Conversion Functions	5-8
Floating-point Math Functions	5-9
Variable Argument Macros	5-10
General Utilities	5-11
String Functions	5-13

Table 5–3. Error Message Macro

(Header File: `assert.h`)

Macro and Syntax	Description

`void assert (expression)`
`int expression;`
	Inserts diagnostic messages into programs

Table 5–4. Character Typing and Conversion Functions

(Header File: `ctype.h`)

Function and Syntax	Description

`int isalnum (c)`
`char c:`
	Identifies alphanumeric-ASCII characters

`int isalpha (c)`
`char c:`
	Identifies alphabetic-ASCII characters

`int isascii (c)`
`char c:`
	Identifies ASCII characters

`int iscntrl (c)`
`char c:`
	Identifies control characters

`int isdigit(c)`
`char c:`
	Identifies numeric characters

`int isgraph (c)`
`char c:`
	Identifies any printing character except a space

`int islower (c)`
`char c:`
	Identifies lowercase alphabetic ASCII characters

`int isprint (c)`
`char c:`
	Identifies printable ASCII characters (including spaces)

Character Typings and Conversion Functions (continued)

Function and Syntax	Description

`int ispunct (c)`
`char c:`
	Identifies ASCII punctuation characters

`int isspace (c)`
`char c:`
	Identifies ASCII spacebar, tab (horizontal or vertical), carriage return, formfeed, and newline characters

`int isupper (c)`
`char c:`
	Identifies uppercase ASCII alphabetic characters

`int isxdigit (c)`
`char c:`
	Identifies hexadecimal digits

`char toascii (c)`
`char c:`
	Masks `c` into a 7-bit ASCII character

`char tolower (c)`
`char c:`
	Converts an uppercase argument to lowercase

`char toupper (c)`
`char c:`
	Converts a lowercase argument to uppercase

Table 5–5. Floating-Point Math Functions

(Header File: `math.h`)

Function and Syntax	Description

`double acos (x)`
`double x;`
	Returns the arc cosine of a floating-point value

`double asin (x)`
`double x;`
	Returns the arc sine of a floating-point value

`double atan (x)`
`double x;`
	Returns the arc tangent of a floating-point value

`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 a floating-point value

`double cosh (x)`
`double x;`
	Returns the hyperbolic cosine of a floating-point value

`double exp (x)`
`double x;`
	Returns the exponential function of a real number

`double fabs (x)`
`double x;`
	Returns the absolute value of a floating-point value

`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`

Floating-Point Math Functions (continued)

Function and Syntax	Description

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

`double` **`ldexp`** `(x, exp)`
`  double x;`
`  int    exp;`
	Multiplies a floating-point number by an integer power of 2

`double` **`log`** `(x)`
`  double x;`
	Returns the natural logarithm of a real number

`double` **`log10`** `(x)`
`  double x;`
	Returns the base-10 logarithm of a real number

`double` **`modf`** `(value, iptr)`
`  double value;`
`  int    *iptr;`
	Breaks a floating-point number 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 a floating-point value

`double` **`sinh`** `(x)`
`  double x;`
	Returns the hyperbolic sine of a floating-point value

`double` **`sqrt`** `(x)`
`  double x;`
	Returns the nonnegative square root of a real number

`double` **`tan`** `(x)`
`  double x;`
	Returns the tangent of a floating-point value

`double` **`tanh`** `(x)`
`  double x;`
	Returns the hyperbolic tangent of a floating-point value

Table 5–6. Variable Argument Macros

(Header File: `stdarg.h`)

Function 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

Table 5–7. General Utilities

(Header File: stdlib.h)

Function and Syntax	Description
int **abs** (j) int j;	Returns the absolute value of an integer
void **abort** ()	Terminates a program abnormally
void **atexit** (fun) void (*fun)();	Registers the function pointed to by fun, to be called without arguments at normal program termination
int **atof** (nptr) char *nptr;	Converts an ASCII string to a floating-point value
int **atoi** (nptr) char *nptr;	Converts an ASCII string to an integer value
long int **atol** (nptr) char *nptr;	Converts an ASCII string 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 a member that matches the object that key points to
void ***calloc** (nmemb, size) size_t nmemb, size	Allocates and clears memory for nmemb objects
void **exit** (status) int status;	Terminates a program normally
void **free** (ptr) void *ptr;	Deallocates memory space allocated by malloc, calloc, or realloc
int **labs** (j) int j;	Returns the absolute value of an integer
int **ltoa** (n, buffer) long n; char *buffer;	Converts a long integer to the equivalent ASCII string
void ***malloc** (size) size_t size	Allocates memory for an object of size bytes
void ***minit**()	Resets the memory pool used for dynamic allocation
char ***movmem** (src,dest,count) char *src, *dest; int count;	Moves count bytes from one address to another
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

General Utilities (continued)

Function and Syntax	Description
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;	Sets the value of seed so that a subsequent call to rand produces a new sequence of pseudo-random numbers
double **strtod** (nptr, endptr) char *nptr, **endptr;	Converts an ASCII string to a floating-point value
long int **strtol** (nptr, endptr, base) char *nptr,**endptr; int base;	Converts an ASCII string to a long integer
unsigned long int **strtoul** **char *nptr, **endptr; int base;**	Converts an ASCII string to a long integer

Table 5–8. 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 an object
int **memcmp** (s1, s2, n) void *s1, *s2; size_t n;	Compares the first n characters of object1 to object2
void ***memcpy** (s1, s2, n) void *s1, *s2; size_t n;	Copies n characters from object1 to object2
void ***memmove** (s1, s2, n) void *s1, *s2; size_t n;	Moves n characters from object1 to object2
void ***memset** (s, c, n) void *s; int c; size_t n;	Copies the value of c into the first n characters of an object
char ***strcat** (s1, s2) char *s1, *s2;	Appends string1 to the end of string2
char ***strchr** (s, c) char *s; int c;	Finds the first occurrence of a character in a string

`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 in the program: <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 a string to a new location

`size_t strcspn (s1, s2)`
`char *s1, *s1;`
	Returns the length of the initial segment of string1 that is entirely made up of characters that are not in string2

`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 string1 to string2

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

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

`char *strpbrk (s1, s2)`
`char *s1, *s2;`
	Locates the first occurrence in string1 of *any* character from string2

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

`size_t strspn (s1, s2)`
`char *s1, *s2;`
	Returns the length of the initial segment of string1 which is entirely made up of characters from string2

`char *strstr (s1, s2)`
`char *s1, *s2;`
	Finds the first occurrence of a string in another string

`char *strtok (s1, s2)`
`char *s1, *s2;`
	Breaks a string into a series of tokens, each delimited by a character from a second string

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:

Function	**Page**
abort	5-16
abs	5-17
acos	5-18
asin	5-19
assert	5-20
atan	5-21
atan2	5-22
atexit	5-23
atof	5-24
atoi	5-24
atan	5-21
bsearch	5-25
calloc	5-26
ceil	5-27
cos	5-28
cosh	5-29
exit	5-30
exp	5-31
fabs	5-32
floor	5-33
fmod	5-34
free	5-35
frexp	5-36
isalnum	5-37
isalpha	5-37
isascii	5-37
iscntrl	5-37
isdigit	5-37
isgraph	5-37
islower	5-37
isprint	
ispunct	5-37
isspace	5-37
isupper	5-37
isxdigit	5-37
ldexp	5-39
log	5-40
log10	5-41
ltoa	5-42
malloc	5-43
memchr	5-44
memcmp	5-45
memcpy	5-46

memmove 5-47
memset 5-48

minit 5-49
modf 5-50
movmem 5-51
pow 5-52
qsort 5-53

rand 5-54
realloc 5-55
sin 5-56
sinh 5-57
sqrt 5-58

srand 5-54
strcat 5-59
strchr 5-60
strcmp 5-61
strcoll 5-61

strcpy 5-62
strcspn 5-63
strerror 5-64
strlen 5-65
strncat 5-66

strncmp 5-67
strncpy 5-68
strpbrk 5-69
strrchr 5-70
strspn 5-71

strstr 5-72
strtod 5-73
strtok 5-74
strtol 5-73
strtoul 5-73

tan 5-75
tanh 5-76
toascii 5-77
tolower 5-78
toupper 5-78

va_arg 5-79
va_end 5-79
va_start 5-79

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.

Because *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`. Argument `x` must be in the range [–1,1]. The return value is an angle in the range [0,π] radians.

Example

```
double realval, radians;

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

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`. Argument `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`. Because 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 [–π,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;
```

Description Three functions convert ASCII strings to numeric representations:

- The **atof** 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 **atoi** function converts a string to an integer. The string must have the following format:

  [space] [sign] digits

- The **atol** function converts a string to a long integer. The string must have the following format:

  [space] [sign] digits

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.

Because *int* and *long int* are functionally equivalent in TMS320C25 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. A C module called `memory.c` reserves memory for the heap in the .bss section. The constant `MEMORY_SIZE` defines the size of the heap as 1000 bytes. If necessary, you can change the size of the heap by changing the value of `MEMORY_SIZE` and recompiling `memory.c`. For more information, refer to Section 4.1.5, Dynamic Memory Allocation, on page 4-5.

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 double-precision 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  <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  <stdlib.h>

void  exit(status)
   int   status;
```

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

Description When a program exits through a call to the exit function, the atexit function calls the registered functions (without arguments) in reverse order of their registration.

The exit function does not return.

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 double-precision 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 4.1.5, Dynamic Memory Allocation, on page 4-5.

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 .78375 and exp is 2 */
```

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 character 0—31 and 127).
isdigit	identifies numeric characters ('0'— '9')
isgraph	identifies any nonspace character.

islower	identifies lowercase alphabetic ASCII characters.
isprint	identifies printable ASCII characters, including spaces (ASCII characters 32—126).
ispunct	identifies ASCII punctuation characters.
isspace	identifies ASCII 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  <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.

Example

```
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. A C module called memory.c reserves memory for the heap in the .bss section. The constant MEMORY_SIZE defines the size of the heap as 1000 bytes. If necessary, you can change the size of the heap by changing the value of MEMORY_SIZE and recompiling memory.c. For more information, refer to Section 4.1.5, Dynamic Memory Allocation, on page 4-5.

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 that the objects memcmp compares can contain zeros.

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 that the objects memcpy copies can contain zeros.

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.

CAUTION

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. A C module called memory.c reserves memory for the heap in the .bss section. The constant `MEMORY_SIZE` defines the size of the heap as 1000 bytes. If necessary, you can change the size of the heap by changing the value of `MEMORY_SIZE` and reassembling `memory.c`. For more information, refer to Section 4.1.5, Dynamic Memory Allocation, on page 4-5.

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)();
```

Fields

`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;
```

Fields `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 */
```

Fields

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

The memory that realloc uses is in a special memory pool or heap. A C module called memory.c reserves memory for the heap in the .bss section. The constant MEMORY_SIZE defines the size of the heap as 1000 bytes. If necessary, you can change the size of the heap by changing the value of MEMORY_SIZE and recompiling memory.c. For more information, refer to Section 4.1.5, Dynamic Memory Allocation, on page 4-5.

Syntax

```
#include  <math.h>

double sin(x)
   double x;
```

Fields

`sin.obj` in `rts.lib`

Description

The sin function returns the sine of a floating-point number, `x`. Argument `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;
```

Fields `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;
```

Fields `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;
```

Fields `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;
```

Fields `strchr.c` in `rts.src`

Description The strchr function finds the first occurrence of `c` (which is first converted to a char) 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;
```

Fields `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;
```

Fields `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 the destination string.

Syntax

```
#include  <string.h>
```

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

Fields `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;
```

Fields `strerror.c` in `rts.src`

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

Syntax

```
#include  <string.h>
```

```
size_t strlen(s)
   char  *s;
```

Fields `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;
```

Fields `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;
```

Fields `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;
```

Fields `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;
```

Fields `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;
```

Fields `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;
```

Fields

`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;
```

Fields `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;
```

Fields

`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 new line. Following the space is an optional *sign*, and then *digits* that represent the integer portion of the number. The fractional part of the number follows, then the exponent, including an optional *sign*.

The first unrecognized character terminates the string. The pointer that `endptr` points to is set to point to this character.

Syntax

```
#include  <string.h>
```

```
char *strtok(s1, s2)
   char *s1, *s2;
```

Fields `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;
```

Fields `tan.obj` in `rts.lib`

Description The tan function returns the tangent of a floating-point number, `x`. Argument `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;
```

Fields `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  <ctype.h>
int  toascii(c)
   char  c;
```

Fields `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 a toascii macro.

Syntax

```
#include  <ctype.h>

int  tolower(c)
   char  c;

int  toupper(c)
   char  c;
```

Fields

`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       */
{
     int  i;
     char *s;
     long l;

     va_list  ap;

     va_start(ap,fmt); /* initialize                          */
         .
         .
         .
                       /* Get next argument, an integer       */
     i = va_arg(ap, int);
                       /* Get next argument, a string         */
     s = va_arg(ap, char *);
                       /* Get next argument, a long           */
     l = va_arg(ap, long);
```

```
                 .
                 .
                 .
            va_end(ap)          /* Reset                              */
        }
```

Appendix A

Fatal Errors

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*

All the errors listed in this section cause the compiler to abort immediately. Text *in italics* in these error messages is replaced with actual text from the program, your own symbols, filenames, memory allocations, etc.

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 by 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 indicate particular errors) and try compilation again.

Too many errors: aborting: An error has occurred that prevents the compiler from continuing.

Cannot recover from earlier errors: aborting: An error has occurred that prevents the compiler from continuing.

Appendix B

C 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 is an alphabetical table of contents for the preprocessor directives reference:

Directive	**Page**
#define	B–2
#else	B–3
#endif	B–3
#if	B–3
#ifdef	B–3
#ifndef	B–3
#include	B–5
#line	B–6
#undef	B–2

Syntax

#define *name* [(arg,...,arg)] token-string

#undef *name*

Description The preprocessor supports two directives for defining and undefining constants:

- The **#define** directive assigns a string to a constant. Subsequent occurrences of *name* (which can be immediately followed by an argument list) 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 constant *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 several 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 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 macro. 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 macro. The code following an #ifndef directive (up to an #else or an #endif) is compiled if *name* is not been defined (by the #define directive) or if it has been 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.

For more information about the –i option and the environment variable, read Section 2.2.1.1 on page 2-8.

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.

Index

A

–al shell program option, 2-3
alternate directories, 2-8
–ap shell program option, 2-3
AR0 (SP), 4-4, 4-6, 4-8
AR1 (FP), 4-4, 4-6, 4-8
AR2, 4-6
archiver, 1-3, 2-20
–as shell program option, 2-4
asm statement, 3-9, 4-14
assembler, 1-3, 2-2, 2-3
assert macro, 5-2
assert.h header, 5-2, 5-8
autoinitialization, 2-17, 4-3, 4-21, 4-22
  RAM model, 2-18, 4-21, 4-22
  ROM model, 2-17, 4-21, 4-22
–ax shell program option, 2-4

B

bit addressing, 4-5
boot.obj, 2-15, 2-17, 2-19, 4-4
.bss section, 4-2, 4-3, 4-4

C

–c linker option, 2-15, 4-3
–c preprocessor option, 2-6
–c shell program option, 2-3
C_DIR environment variable, 2-9
c_int0, 2-16
calloc function, 4-5
character typing conversion functions
  isalnum, 5-37
  isalpha, 5-37
  isascii, 5-37
  iscntrl, 5-37
  isdigit, 5-37
  isgraph, 5-37
  islower, 5-37
  isprint, 5-37
  ispunct, 5-37
  isspace, 5-37
  isupper, 5-37
  isxdigit, 5-37
  toascii, 5-77
  tolower, 5-78
  toupper, 5-78
character typing/conversion functions, 5-3, 5-8
.cinit, 2-17
.cinit section, 4-2, 4-20, 4-21
code generator, 2-5
  dspcg, 2-13
  invocation, 2-13
  options
    –o, 2-13
    –q, 2-14
    –z, 2-14
compiler operation, 2-1—2-20
–cr linker option, 2-15, 4-4
ctype.h header, 5-3, 5-8

D

–d preprocessor option, 2-6
.data section, 4-3
data types, 3-5
  data sizes, 3-5
dedicated registers, 4-6
#define directive, B-2
derived types, 3-4
diagnostic messages, 5-2
  assert, 5-20
  NDEBUG macro. *See* NDEBUG macro
div_t type, 5-7
division and modulus, 4-18
–dNAME shell program option, 2-3
dspcc, 2-11
dspcg, 2-13
dspcl, 2-3
dspcpp, 2-6
dynamic memory allocation, 4-5

E

#else directive, B-3
emulator, 1-3
#endif directive, B-3
entry points
  C code, 2-16—2-20
  _c_int0, 2-16
  reset vector, 2-16
enum, 3-7
enumeration declaration, 3-7
environment variable, 2-9
EPROM programmer, 1-3
error message macros, 5-8
  assert, 5-20
error messages, A-1
expression analysis
  floating point, 4-19
  integers, 4-18
external definitions, 3-7
external variables, 4-3

F

fatal errors, A-1
field manipulation, 4-5
float.h header, 5-3
floating-point expression analysis, 4-19
floating-point math functions, 5-9
  acos, 5-18
  asin, 5-19
  atan, 5-21
  atan2, 5-22
  ceil, 5-27
  cos, 5-28
  cosh, 5-29
  exp, 5-31
  fabs, 5-32
  floor, 5-33
  fmod, 5-34
  frexp, 5-36
  ldexp, 5-39
  log, 5-40
  log10, 5-41
  modf, 5-50
  pow, 5-52
  sin, 5-56
  sinh, 5-57
  sqrt, 5-58
  tan, 5-75
  tanh, 5-76
FP register, 4-4, 4-6, 4-8
frame pointer, 4-6, 4-8
function call conventions, 4-8—4-22
  local frame generation, 4-8
  passing parameters, 4-8
  terminating a function, 4-9
  termination, 4-9
function calls, using the stack, 4-4

G

–g shell program option, 2-3
general utility functions, 5-7, 5-11
  abort, 5-16
  abs, 5-17
  atexit, 5-23

bsearch, 5-25
calloc, 5-26
exit, 5-30
free, 5-35
labs, 5-17
malloc, 5-43
minit, 5-49
movmem, 5-51
qsort, 5-53
rand, 5-54
realloc, 5-55
srand, 5-54
strtod, 5-73
strtol, 5-73
strtoul, 5-73
global variables, 3-9, 4-3
reserved space, 4-2

H

hardware requirements (PC systems), 1-4
header files, 5-2—5-7
assert.h header, 5-2
ctype.h header, 5-3
float.h header, 5-3
limits.h header, 5-3
math.h header, 5-5
stdarg.h header, 5-6
stddef.h header, 5-6
stdlib.h header, 5-7
string.h header, 5-7
heap, 4-5

I

–i preprocessor option, 2-7, 2-8
–i shell program option, 2-3
identifiers, 3-2
#if directive, B-3
#ifdef directive, B-3
#ifndef directive, B-3
#include, 2-8
#include directive, B-5
includes, 2-8
initialized sections, 4-2
inline assembly construct (asm), 3-9, 4-14
installation, 1-4
Macintosh, 1-4
MPW, 1-4, 1-7
OS/2, 1-4
PC, 1-4
PC-DOS, 1-4
Sun-3 workstation, 1-4
Sun-OS, 1-4, 1-6
Ultrix, 1-4, 1-6
VAX, 1-4
VMS, 1-4, 1-5
integer expression analysis, 4-18
division and modulus, 4-18
overflow and underflow, 4-18
interfacing C and assembly language, 4-11—4-15
asm statement, 4-14
assembly language modules, 4-11
defining variables in assembly language, 4-13
inline assembly language, 4-14
modifying compiler output, 4-15
interrupt handling, 4-16
invoking the
assembler, 1-4
batch files, 1-4
code generator, 2-5
linker, 2-15
parser, 2-5
preprocessor, 2-5
invoking the assembler, 2-2

K

–k shell program option, 2-3
Kernighan and Ritchie
preprocessor, 2-7
The C Programming Language, 1-1
keywords, 3-2

L

–l linker option, 2-15
ldiv_t type, 5-7

limits
  floating-point types, 5-3, 5-5
  integer types, 5-3
limits.h header, 5-3
#line directive, B-6
linker, 1-3, 2-15—2-20
linker command file, 2-16
linking C code, 2-15—2-20
local frame generation, 4-8
local variable pointer, 4-6

M

–ma shell program option, 2-3
Macintosh installation, 1-7
malloc function, 4-5
math.h header, 5-5, 5-9
  directives
    #define, B-2
    #else, B-3
    #endif, B-3
    #if, B-3
    #ifdef, B-3
    #ifndef, B-3
    #include, B-5
    #line, B-6
    #undef, B-2
  floating-point math functions, 5-5
memory management functions
  calloc, 5-26
  free, 5-35
  malloc, 5-43
  minit, 5-49
  movmem, 5-51
  realloc, 5-55
memory model, 4-2—4-22
  allocating variables, 4-3
  dynamic memory allocation, 4-5
  field manipulation, 4-5
  RAM model, 4-3
  ROM model, 4-3
  runtime stack, 4-4
  sections, 4-2
  structure packing, 4-5
memory.c, 4-5
MEMORY_SIZE constant, 4-5
modulus, 4-18
–mr shell program option, 2-3
–mv shell program option, 2-3

N

–n shell program option, 2-3
name types, 3-4
NDEBUG macro, 5-2
NULL macro, 5-6

O

–o code generator option, 2-13
–o linker option, 2-15
object alignment, 3-6
object format converter, 1-3
object libraries, 2-15
offsetof macro, 5-6
operation of the compiler, 2-1—2-20
overflow, 4-18

P

–p preprocessor option, 2-7
packing structures, 4-5
parser (dspcc), 2-11
  invocation, 2-11
  options, 2-12
    –q, 2-12
    –z, 2-12
passing parameters to a function, 4-8
PC installation, 1-4
–pc shell program option, 2-3
pointers, 3-5
  data size, 3-5
–pp shell program option, 2-3
predefined names, 2-7
preprocessor (dspcpp), 2-6, B-2
  invocation, 2-6
  options, 2-3, 2-6

–c, 2-6
–d, 2-6
–i, 2-7
–p, 2-7
–q, 2-7

preprocessor directives, B-1

program termination functions
abort (exit), 5-16
atexit, 5-23
exit, 5-30

ptrdiff_t type, 5-6

Q

–q code generator option, 2-14
–q parser option, 2-12
–q preprocessor option, 2-7
–q shell program option, 2-3
–qq shell program option, 2-3

R

RAM model of autoinitialization, 2-15—2-20, 4-3
RAND_MAX macro, 5-7
realloc function, 4-5
register conventions, 4-6—4-22
dedicated registers, 4-6
register variables, 4-7
register variables, 3-9, 4-7
reserved registers, 4-6
reset vector, 2-16
ROM model of autoinitialization, 2-15—2-20, 4-3
rts.lib, 2-15—2-20, 5-1
rts.src, 4-4, 5-1, 5-7
runtime environment, 4-1—4-22
floating-point expression analysis, 4-19
function call conventions, 4-8—4-22
integer expression analysis, 4-18
interfacing C with assembly language, 4-11
assembly language modules, 4-11
defining variables in assembly language, 4-13
inline assembly language, 4-14
modifying compiler output, 4-15
interrupt handling, 4-16
memory model, 4-2—4-22
allocating variables, 4-3
dynamic memory allocation, 4-5
field manipulation, 4-5
RAM model, 4-3
ROM model, 4-3
sections, 4-2
structure packing, 4-5
register conventions, 4-6—4-22
runtime stack, 4-4
system initialization, 4-20—4-22
runtime initialization, 2-15
runtime stack, 4-20
runtime support, 2-15
runtime-support functions, 5-1—5-15
descriptions, 5-14—5-15
summary table, 5-8—5-15

S

–s shell program option, 2-3
sections, 4-2
.bss, 4-2, 4-3, 4-4
.cinit, 4-2, 4-3, 4-20, 4-21
.data, 4-3
.text, 4-3
shell program, options
–al, 2-3
–ap, 2-3
–as, 2-4
–ax, 2-4
–c, 2-3
–dNAME, 2-3
–i, 2-3
–k, 2-3
–ma, 2-3
–mr, 2-3
–mv, 2-3
–n, 2-3
–pc, 2-3
–pp, 2-3

–q, 2-3
–qq, 2-3
–s, 2-3
–uNAME, 2-3
–z, 2-3
simulator, 1-3
size_t type, 5-6
software development system, 1-3
software installation, 1-4
SP register, 4-4, 4-6, 4-8
stack, 4-4
stack management, 4-4
stack pointer, 4-4, 4-6, 4-8
STACK_SIZE constant, 4-4
static variable, 3-9
static variables, 4-3
reserved space, 4-2
stdarg.h header, 5-6, 5-10
stddef.h header, 5-6
stdlib.h header, 5-7, 5-11
storage class specifiers, 3-6
string functions, 5-7, 5-12
memchr, 5-44
memcmp, 5-45
memcpy, 5-46
memmove, 5-47
memset, 5-48
strcat, 5-59
strchr, 5-60
strcmp, 5-61
strcoll, 5-61
strcpy, 5-62
strcspn, 5-63
strerror, 5-64
strlen, 5-65
strncat, 5-66
strncmp, 5-67
strncpy, 5-68
strpbrk, 5-69
strrchr, 5-70
strspn, 5-71
strstr, 5-72
strtok, 5-74
string.h header, 5-7, 5-12
structure packing, 4-5
Sun-3 installation, 1-6
system constraints
MEMORY_SIZE, 4-5
STACK_SIZE, 4-4
system initialization, 4-20—4-22
autoinitialization, 4-21
runtime stack, 4-20
system stack, 4-4
_ _SYSTEM_SIZE, 5-7

T

–t code generator option, 2-14
.text section, 4-2
type specifiers, 3-7

U

–uNAME shell program option, 2-3
#undef directive, B-2
underflow, 4-18
uninitialized sections, 4-2

V

variable allocation, 4-3
variable argument functions, 5-6
variable argument functions and macros
va_arg, 5-79
va_end, 5-79
va_start, 5-79
variable argument macros, 5-10
VAX installation, 1-5, 1-6

X

XDS emulator, 1-3

Z

z code generator option, 2-14
–z parser option, 2-12
–z shell program option, 2-3

TEXAS
INSTRUMENTS

Printed in U.S.A., December 1989
1604909-9707

SPRU024A