Texas Instruments

TMS34010 C Compiler

Reference Guide

1988

Graphics Products

TMS34010 C Compiler Reference Guide

TEXAS INSTRUMENTS

IMPORTANT NOTICE

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

TI warrants performance of its semiconductor products to current specifications in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Unless mandated by government requirements, specific testing of all parameters of each device is not necessarily performed.

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

Contents

Section		*Page*
1	**Introduction**	**1-1**
1.1	Software Development Tools Overview	1-2
1.2	Related Documentation	1-4
1.3	Style and Symbol Conventions	1-5
1.4	Getting Started	1-6
2	**Compiler Installation**	**2-1**
2.1	Installing the C Compiler on IBM/TI PCs with PC/MS-DOS	2-2
2.2	Installing the C Compiler on VAX/VMS	2-3
2.3	Installing the C Compiler on UNIX Systems	2-4
2.4	Installing the C Compiler on Macintosh/MPW Systems	2-5
3	**C Compiler Operation**	**3-1**
3.1	Preprocessor (gspcpp) Description	3-2
3.1.1	Invoking the C Preprocessor	3-2
3.1.2	General Information	3-3
3.1.3	Specifying Alternate Directories for Include Files	3-4
3.2	Parser (gspcc) Description	3-6
3.2.1	Invoking the Parser	3-6
3.2.2	General Information	3-7
3.3	Code Generator (gspcg) Description	3-8
3.3.1	Invoking the Code Generator	3-8
3.3.2	Pointers to Named Variables (-a Option)	3-9
3.3.3	Small Code Model (-s Option)	3-10
3.3.4	Checking for Stack Overflow (-x option)	3-10
3.4	Compiling and Assembling a Program	3-11
3.5	Linking a C Program	3-13
3.5.1	Runtime Initialization and Runtime Support	3-13
3.5.2	Sample Linker Command File	3-14
3.5.3	Autoinitialization (ROM and RAM Models)	3-15
3.5.4	The -c and -cr Linker Options	3-15
3.6	Archiving a C Program	3-16
4	**The TMS34010 C Language**	**4-1**
4.1	Identifiers, Keywords, and Constants	4-2
4.2	TMS34010 C Data Types	4-4
4.3	Object Alignment	4-6
4.4	Conversions	4-6
4.5	Expressions	4-7
4.6	Declarations	4-8
4.7	Initialization of Static and Global Variables	4-10
4.8	asm Statement	4-10
4.9	Lexical Scope Rules	4-11

5 Runtime Environment 5-1
5.1 Memory Model . . . 5-2
5.1.1 Sections . . . 5-2
5.1.2 Stack Management . . . 5-3
5.1.3 Dynamic Memory Allocation . . . 5-4
5.1.4 RAM and ROM Models . . . 5-4
5.1.5 Allocating Memory for Static and Global Variables . . . 5-5
5.1.6 Packing Structures and Manipulating Fields . . . 5-5
5.1.7 Array Alignment . . . 5-5
5.2 Register Conventions . . . 5-6
5.2.1 Dedicated Registers . . . 5-6
5.2.2 Using Registers . . . 5-6
5.2.3 Register Variables . . . 5-7
5.3 Function Structure and Calling Conventions . . . 5-8
5.3.1 Responsibilities of a Calling Function . . . 5-8
5.3.2 Responsibilities of a Called Function . . . 5-9
5.3.3 Setting up the Local Frame . . . 5-10
5.3.4 Accessing Arguments and Local Variables . . . 5-10
5.3.5 Returning Structures from Functions . . . 5-11
5.4 Interfacing C with Assembly Language . . . 5-12
5.4.1 Assembly Language Modules . . . 5-12
5.4.2 Inline Assembly Language . . . 5-15
5.4.3 Modifying Compiler Output . . . 5-15
5.5 Interrupt Handling . . . 5-16
5.6 Integer Expression Analysis . . . 5-17
5.7 Floating-Point Support . . . 5-17
5.7.1 Floating-Point Formats . . . 5-17
5.7.2 Double-Precision Functions . . . 5-19
5.7.3 Single-Precision Functions . . . 5-20
5.7.4 Conversion Functions . . . 5-21
5.7.5 Floating-Point Errors . . . 5-21
5.8 System Initialization . . . 5-22
5.8.1 Initializing the Stack . . . 5-22
5.8.2 Autoinitialization of Variables and Constants . . . 5-23

6 Runtime-Support Functions 6-1
6.1 Header Files . . . 6-2
6.1.1 Diagnostic Messages (assert.h) . . . 6-2
6.1.2 Character Typing and Conversion (ctype.h) . . . 6-3
6.1.3 Limits (float.h and limits.h) . . . 6-3
6.1.4 Floating-Point Math (math.h, errno.h) . . . 6-5
6.1.5 Nonlocal Jumps (setjmp.h) . . . 6-5
6.1.6 Variable Arguments (stdarg.h) . . . 6-6
6.1.7 Standard Definitions (stddef.h) . . . 6-6
6.1.8 General Utilities (stdlib.h) . . . 6-6
6.1.9 String Functions (string.h) . . . 6-7
6.1.10 Time Functions (time.h) . . . 6-8
6.2 Summary of Runtime-Support Functions and Macros . . . 6-9
6.3 Functions Reference . . . 6-14

A Error Messages A-1
B C Preprocessor Directives B-1

Illustrations

Figure *Page*

1-1 TMS34010 Software Development Flow 1-2
3-1 Compiling a C Program 3-1
3-2 Input and Output Files for the C Preprocessor 3-2
3-3 Input and Output Files for the C Parser 3-6
3-4 Input and Output Files for the C Code Generator 3-8
3-5 An Example of a Linker Command File 3-14
5-1 The Program and System Stacks 5-3
5-2 An Example of a Function Call 5-8
5-3 Single-Precision Format 5-18
5-4 Double-Precision Format 5-18
5-5 Format of Initialization Records in the .cinit Section 5-24
5-6 ROM Model of Autoinitialization 5-25
5-7 RAM Model of Autoinitialization 5-26

Tables

Table *Page*

6-1 Macros that Supply Integer Type Range Limits (limits.h) 6-3
6-2 Macros that Supply Floating-Point Range Limits (float.h) 6-4

Preface

The *TMS34010 C Compiler Reference Guide* contains the following sections:

Section 1 Introduction

Overviews the TMS34010 development tools and the code development process, lists related documentation, describes style and symbol conventions used in this document, and provides a walkthrough.

Section 2 Software Installation

Contains instructions for installing the C compiler on VAX/VMS, VAX/Ultrix, VAX/System V, IBM-PC/PC-DOS, and TI-PC/MS-DOS systems.

Section 3 Compiler Operation

Describes the three major components of the C compiler (preprocessor, parser, and code generator), contains instructions for invoking these components individually or for invoking batch files to compile and assemble a C source file, discusses linking C programs, and discusses archiving C programs.

Section 4 TMS34010 C Language

Discusses the differences between the C language supported by the TMS34010 C compiler and standard Kernighan and Ritchie C.

Section 5 Runtime Environment

Contains technical information on how the compiler uses the TMS34010 architecture; discusses memory and register conventions, stack organization, function-call conventions, and system initialization; provides information needed for interfacing assembly language to C programs.

Section 6 Runtime-Support Functions

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

Appendix A Error Messages

Shows the format of compiler error messages and lists all the error messages that are fatal.

Appendix B Preprocessor Directives

Describes the standard preprocessor directives that the compiler supports.

Section 1

Introduction

The TMS34010 Graphics System Processor is an advanced 32-bit microprocessor optimized for graphics systems. The TMS34010 is a member of the TMS340 family of computer graphics products from Texas Instruments.

The TMS34010 is well supported by a full set of hardware and software development tools, including a C compiler, a full-speed emulator, a software simulator, and an IBM/TI-PC development board. (Section 1.1 describes these tools.)

This reference guide describes the TMS34010 C compiler. Its main purpose is to present the details and characteristics of this particular C compiler; it assumes that you already know how to write C programs. We suggest that you obtain a copy of *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie (published by Prentice-Hall); use this reference guide as a supplement to the Kernighan and Ritchie book.

The TMS34010 C compiler can be installed on the following systems:

- **PCs:**
  - IBM-PC with PC-DOS
  - TI-PC with MS-DOS
- **VAX:**
  - VMS
  - Ultrix
- **Apollo Workstations:**
  - Domain/IX
  - AEGIS
- **Sun-3 Workstations** with Unix
- **Macintosh** with MPW

Topics in this introductory section include:

Section		Page
1.1	Software Development Tools Overview	1-2
1.2	Related Documentation	1-4
1.3	Style and Symbol Conventions	1-5
1.4	Getting Started	1-6

1.1 Software Development Tools Overview

Figure 1-1 illustrates the TMS34010 software development flow. The center portion of the figure highlights the most common path of software development; the other portions are optional.

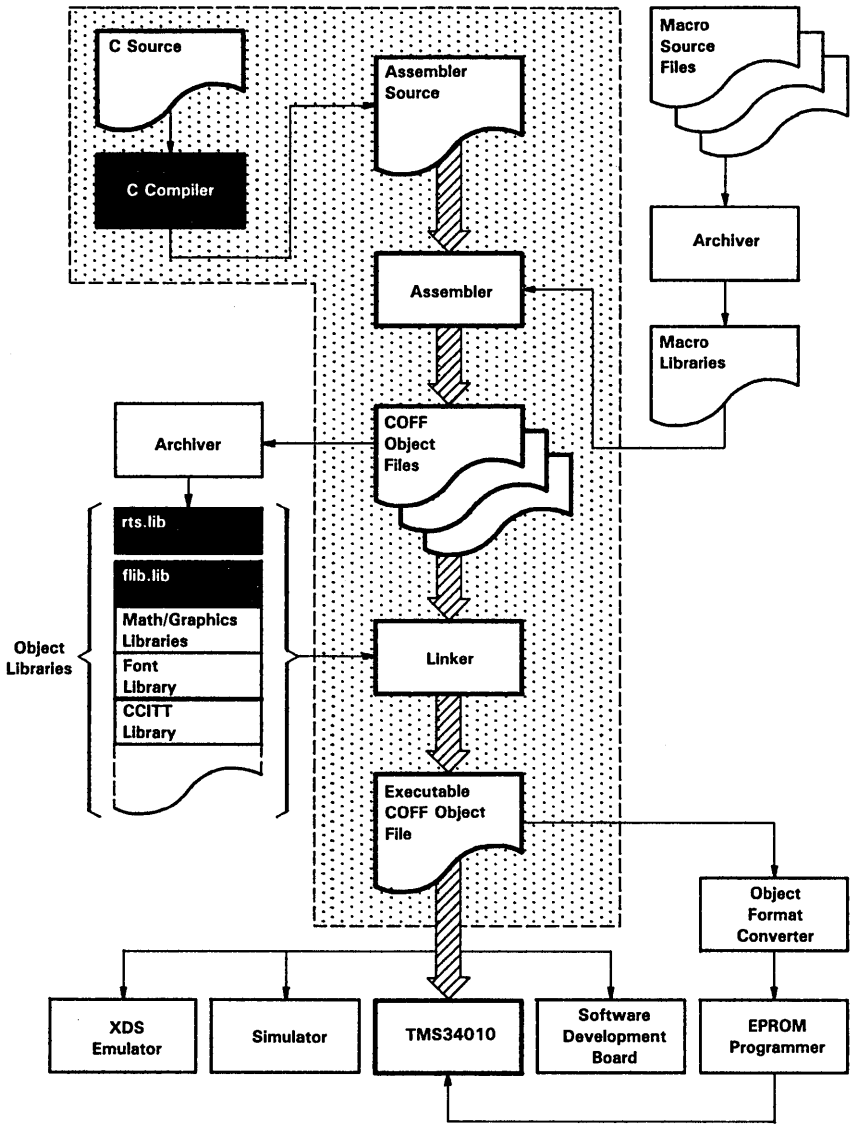

Figure 1-1. TMS34010 Software Development Flow

The following list describes the tools that are shown in Figure 1-1.

- The **C compiler** accepts C source code and produces TMS34010 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 and a source library are included with the C compiler:

  - **flib.lib** contains floating-point arithmetic routines.
  - **rts.lib** contains standard runtime-support functions.
  - **rts.src** contains the *source* for the functions in `rts.lib`.

  Several application-specific object libraries are available as separate GSP products:

  - The **math/graphics function library** contains math functions for performing algebraic, trigonometric, and transcendental operations as well as graphics functions for performing viewport management, bit-mapped text, graphics output, color-palette control, three-dimensional transformations, and graphics initialization.
  - The **font library** contains a variety of proportionally spaced and monospaced fonts. You can use the functions in the graphics library to display the fonts.
  - The **CCITT data compression function library** contains CCITT-compatible routines for compressing and decompressing monochrome image data.
  - The **8514 adaptor emulation function library** contains routines for use with the IBM PS/2 high-resolution display.

  These functions and routines can be called from 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 any functions called from a C program.

- The **linker** combines object files into a single executable object module. As the linker 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 **TMS34010 target system**. You can use one of several debugging tools to refine and correct your code. Available products include: a software **simulator** that runs on PCs, a PC-based **software development board** (SDB), and a realtime in-circuit **XDS/22 emulator**.

- 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 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 TMS34010 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 ***TMS34010 Assembly Language Tools User's Guide*** (literature number SPVU004) tells you how to use the TMS34010 assembler, linker, archiver, object format converter, and simulator.
- The ***TMS34010 Math/Graphics Function Library User's Guide*** (literature number SPVU006) describes a collection of mathematics and graphics functions that can be called from C programs.
- The ***TMS34010 CCITT Data Compression Function Library User's Guide*** (literature number SPVU009) describes a collection of CCITT-compatible routines for compressing and decompressing monochrome image data.
- The ***TMS34010 Font Library User's Guide*** (literature number SPVU007) describes a set of fonts that are available for use in a TMS34010-based graphics system.
- The ***TMS34010 User's Guide*** (literature number SPVU001) discusses hardware aspects of the TMS34010 such as pin functions, architecture, stack operation, and interfaces, and contains the TMS34010 instruction set.
- The ***TMS34010 Application Guide*** (literature number SPVA007) is a collection of individual application reports. Each report pertains to a specific TMS34010 application. Typical applications discuss topics such as using a TMS34010 in a 512×512-pixel minimum-chip system, designing TMS34010-based systems that are compatible with various graphics standards, and interfacing the TMS34010 to a variety of host processors.
- The ***TMS34010 Software Development Board User's Guide*** (literature number SPVU002) describes using the TMS34010 software development board (a high-performance, PC-based graphics card) for testing and developing TMS34010-based graphics systems.

1.3 Style and Symbol 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:

```
#include    <memory.h>
   int  free(pointer)
   char *pointer;
```

  Some examples show screen displays in the special font; the part of the display that *you enter* is shown in the **bold special font**. In the following example, *you* enter the first line to invoke the parser; the next three lines are messages that the parser prints to the screen.

```
gspcc program
C Compiler,                                Version 3.xx
(c) Copyright 1988, Texas Instruments Incorporated
   "program.c"  ==> main
```

- In syntax descriptions, the instruction, command, or directive is in a **bold face font**. Parameters are in *italics*. Here is an example of directive syntax:

  #line *integer-constant [" filename"]*

  #line is a preprocessor directive. This directive has two parameters, indicated by *integer-constant* and *"filename"*. When you use #line, the first parameter must be an actual integer constant; the second parameter must be the name of a file, enclosed in double quotes.

- Square brackets ([ ]) indicate an optional parameter. Here's an example of a command that has three optional parameters:

  gspcpp *[input file [output file]] [options]*

  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 aren't optional).

1.4 Getting Started

The TMS34010 C compiler has three parts: a preprocessor, a parser, and a code generator. The compiler produces an assembly language source file that must be assembled and linked. The simplest way to compile and assemble a C program is to use the **gspc** batch file which is included with the compiler. This section provides a quick walkthrough so that you can get started without reading the entire reference guide.

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

```
/*********************************/
/*          function.c           */
/* (Sample file for walkthrough) */
/*********************************/
#include "stdlib.h"
int abs(i)
 int i;
{
 register int temp = i;
 if (temp < 0) temp *= -1;
 return (temp);
}
```

2) Invoke the `gspc.bat` batch file to compile and assemble `function.c`; enter:

```
gspc function
```

The **gspc** command invokes the batch file, which in turn invokes the C preprocessor, C parser, C code generator, and the 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 an extension of .c.

After you invoke the batch file, it will print the following progress messages:

```
---[function]---
C Pre-Processor,        Version 3.xx
(c) Copyright 1988, Texas Instruments Incorporated
C Compiler,             Version 3.xx
(c) Copyright 1988, Texas Instruments Incorporated
   "function.c" ==> abs
C Codegen,              Version 3.xx
(c) Copyright 1988, Texas Instruments Incorporated
   "function.c" ==> abs
COFF Assembler,         Version 3.xx
(c) Copyright 1988, Texas Instruments Incorporated
 PASS 1
 PASS 2
No Errors, No Warnings
       Successful Compile of Module function
```

Each component of the compiler creates a file that the next component uses as input (for example, the preprocessor creates an input file for the parser). Each component names its output file by using the source filename with special extensions that indicate which component created the file.

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 of the batch file 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`:

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

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

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

Section		Page
3.1	Preprocessor (gspcpp) Description	3-2
3.2	Parser (gspcc) Description	3-6
3.3	Code Generator (gspcg) Description	3-8
3.4	Compiling and Assembling a Program	3-11
3.5	Linking a C Program	3-13

Section 2

Compiler Installation

This section contains step-by-step instructions for installing the TMS34010 C compiler. The compiler can be installed on the following systems:

- **DOS Systems**
  - IBM-PC with PC-DOS[1] (versions 2.1 and up)
  - TI-PC with MS-DOS[2] (versions 2.1 and up)
- **UNIX[3] Systems**
  - VAX/Ultrix
  - Apollo Domain/IX
  - Apollo AEGIS
  - Sun-3
- **DEC VAX/VMS[4]**
- **Apple Macintosh/MPW[5]**

You will find the installation instructions for these systems in the following sections:

Section		Page
2.1	PC Installations	2-2
2.2	VAX/VMS Installation	2-3
2.3	UNIX Systems Installation	2-4
2.4	Macintosh/MPW Installation	2-5

> **Note:**
>
> In order to use the TMS34010 C compiler, you must also have the TMS34010 assembler and linker.

---

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

2 MS-DOS is a trademark of Microsoft Corporation.

3 UNIX is a registered trademark of AT&T.

4 VAX and VMS are trademarks of Digital Equipment Corporation.

5 Macintosh and MPW are trademarks of Apple Computer, Inc.

2.1 Installing the C Compiler on IBM/TI PCs with PC/MS-DOS

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

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

A: Floppy-disk drive for hard disk systems; source drive for dual-drive systems.
B: Destination or system disk for dual-drive systems.
C: Winchester (hard disk) for hard-disk systems. (**E:** on TI PCs.)

Follow these instructions to install the software:

1) Make backups of the product diskettes.

2) Create a directory to contain the C compiler. If you're using a dual-drive system, put the disk that will contain the tools into drive B.

   - On *hard-disk* systems, enter:

     ```
     MD C:\GSPTOOLS
     ```

   - On *dual-drive* systems, enter:

     ```
     MD B:\GSPTOOLS
     ```

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

   - On *hard-disk* systems, enter:

     ```
     COPY A:\*.* C:\GSPTOOLS\*.*
     ```

   - On *dual-drive* systems, enter:

     ```
     COPY A:\*.* B:\GSPTOOLS\*.*
     ```

4) Repeat steps 1 through 3 for each product diskette.

2.2 Installing the C Compiler on VAX/VMS

The TMS34010 C compiler tape was created with the VMS BACKUP utility at 1600 BPI. These tools were developed on version 4.5 of VMS. If you are using an earlier version of VMS, you may need to relink the object files; refer to the Release Notes for relinking instructions.

Follow these instructions to install the compiler:

1) Mount the tape on your tape drive.

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

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

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

```
$ @setup DEST:directory
```

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

You may want to include the commands from `setup.com` in your `login.com` file. This automatically defines symbols for running the tools each time you log in.

2.3 Installing the C Compiler on UNIX Systems

The TMS34010 C compiler product tape was made at 1600 BPI using the *tar* utility. Follow these instructions to install the compiler:

1) Mount the tape on your tape drive.

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

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

```
tar  x
```

This copies the entire tape into the directory.

> **Note to Apollo Users:**
>
> These tools can run under either the AEGIS system or Domain/IX. However, when you install the tools, you must use Domain/IX because the tape is in tar format and only Domain/IX has a tar command. If you are not accustomed to using Domain/IX, you can run the tools under AEGIS after they are installed.

2.4 Installing the C Compiler on Macintosh/MPW Systems

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

- *Tools,*
- *Includes,* **and**
- *Libraries.*

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 run-time-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 *Include* directory into a new folder, and use the C—DIR environment variable (see Section 3.1.3 on page 3-4) to create a path to this folder.

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.

Section 3

C Compiler Operation

Figure 3-1 illustrates the three-step process of compiling a C program.

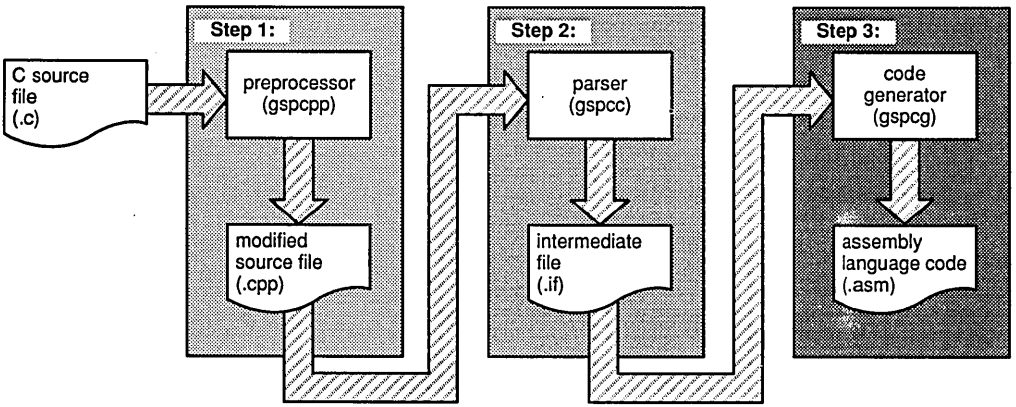

Figure 3-1. Compiling a C Program

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.

After you compile a program, you must assemble and link it with the TMS34010 assembler and linker.

Topics in this section include:

Section		Page
3.1	Preprocessor (gspcpp) Description	3-2
3.2	Parser (gspcc) Description	3-6
3.3	Code Generator (gspcg) Description	3-8
3.4	Compiling and Assembling a Program	3-11
3.5	Linking a C Program	3-13
3.6	Archiving a C Program	3-16

3.1 Preprocessor (gspcpp) Description

The first step in compiling a TMS34010 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 can be used as input for the C parser.

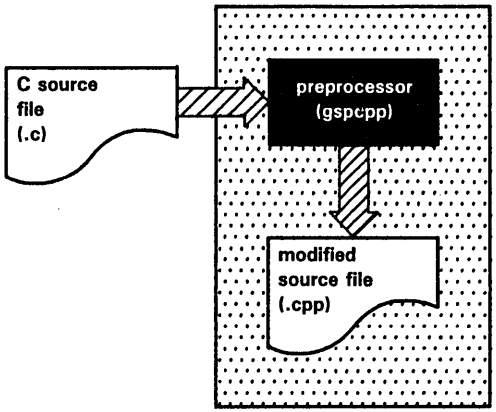

Figure 3-2. Input and Output Files for the C Preprocessor

3.1.1 Invoking the C Preprocessor

To invoke the preprocessor, enter:

gspcpp *[input file [output file]] [options]*

gspcpp is the command that invokes the preprocessor.

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

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

options affect the way the preprocessor processes your input file. An option is a single letter preceded by a hyphen; some options have additional fields which follow the option with no intervening spaces. Options are not case sensitive. Valid options include:

- **-c** copies comments to the output file. If you don't use this option, the preprocessor strips comments. 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 this 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. (See Section 3.1.3, page 3-4.)

-p prevents the preprocessor from producing line number and file information.

-q is the "quiet" option; it suppresses the banner and status information.

Note that options can appear anywhere on the command line.

3.1.2 General Information

- This preprocessor is the same preprocessor that is described in Kernighan and Ritchie; additional information can be found in that book. This 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 and tabs may appear between the # sign and the directive name.

- The C preprocessor maintains and recognizes five predefined macro names:

  `__LINE__` represents the current line number (maintained as a decimal integer).

  `__FILE__` represents the current filename (maintained as a C string).

  `__DATE__` represents the date that the module was compiled (maintained as a C string).

  `__TIME__` represents the time that this module was compiled (maintained as a C string).

  `_gspc` identifies the compiler as the TMS34010 C compiler; this symbol is defined as the constant *1*.

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

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

  would translate into a line such as:

  ```
  printf(%s %s", "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.

3.1.3 Specifying Alternate Directories for Include Files

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

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

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

- If you provide path information for *filename*, the preprocessor uses that path and *does not look* for the file in any other directories.
- If you do not provide path information and you enclose the *filename* in **angle brackets**, the preprocessor searches for the file in:

  1) Any directories named with the -i preprocessor option.
  2) Any directories set with the environment variable C_DIR.

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

- 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 set with the environment variable C_DIR.

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

3.1.3.1 -i Preprocessor Option

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

gspcpp -i*pathname*

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

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

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

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

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

3.1.3.2 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>
```

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

	Pathnames for `alt1.c` and `alt2.c`	Invocation Command
DOS:	`c:\gsp\files\alt1.c` `c:\gsys\alt2.c`	`set C_DIR=c:\gsys c:\exec\files` `gspcpp -ic:\gsp\files source.c`
VMS:	`[gsp.files]alt1.c` `[gsys]alt2.c`	`assign C_DIR "[gsys] [exec.files]"` `gspcpp -i[gsp.files] source.c`
UNIX:	`/gsp/files/alt1.c` `/gsys/alt2.c`	`setenv C_DIR "/gsys /exec/files` `gspcpp -i\gsp\files source.c`

MPW:	`:gsp :files :alt1 .c`
	`:gsys :alt2 .c`
		`set C_DIR " :gsys :files "`
		`export C_DIR`
		`gspcpp -i:gsp :files source.c`

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

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

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

3.2 Parser (gspcc) Description

The second step in compiling a TMS34010 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.) Figure 3-3 illustrates this process.

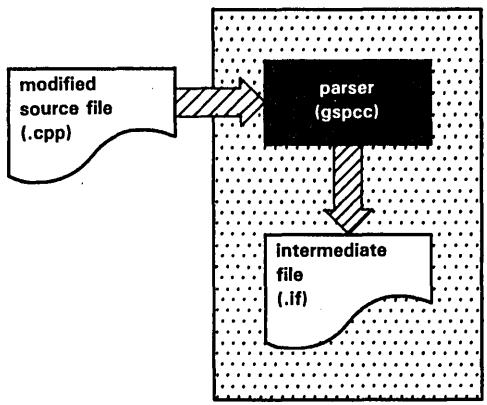

Figure 3-3. Input and Output Files for the C Parser

3.2.1 Invoking the Parser

To invoke the parser, enter:

gspcc *[input file [output file]] [options]*

gspcc 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 will prompt you for one.

output file names the intermediate file that the parser creates. If you don't supply a filename for the output file, the parser uses the input filename with an extension of ***.if***.

options affect the way the parser processes the input file. An option is a single letter preceded by a hyphen. Options can appear anywhere on the command line and are not case sensitive. Valid options include:

-q is the "quiet" option; it suppresses the banner and status information.

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

3.2.2 General Information

- Most errors are fatal; that is, they prevent the parser from generating an intermediate file and must be corrected before you can finish compiling a program. Some errors, however, merely produce warnings which hint of problems but do not 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 an error. You can use the -q option to suppress these messages.

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

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

3.3 Code Generator (gspcg) Description

The third step in compiling a TMS34010 C program is invoking the C code generator. As Figure 3-4 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 TMS34010 assembler. The code generator produces reentrant relocatable code which, after assembling and linking, can be stored in ROM.

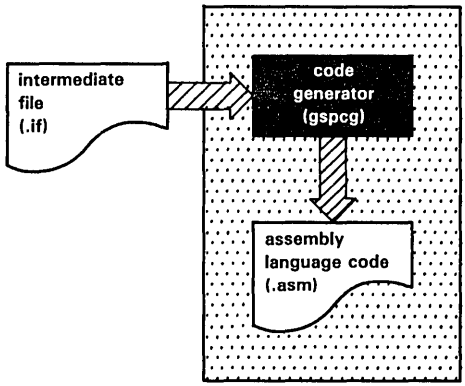

Figure 3-4. Input and Output Files for the C Code Generator

3.3.1 Invoking the Code Generator

To invoke the code generator, enter:

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

gspcg is the command that invokes the code generator.

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

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

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

options affect the way the code generator processes the input file. An option is a single letter preceded by a hyphen. Options can appear anywhere on the command line and are not case sensitive. Valid options include:

-a indicates that the program may contain assignments in the form `*ptr = ...`, where `ptr` is a pointer to a named variable. (See Section 3.3.2 below.)

-o places high-level-language debugging directives in the output file. See Appendix B of the *TMS34010 Assembly Language Tools User's Guide* for more information about these directives.

-q is the "quiet" option; it suppresses the banner and status information.

-r periodically writes a register-status table to the output file. This table is a list of assembly language comments that names each register that the code generator is currently using; it also shows the type of each register's current contents. The table is printed between statements whenever the contents of registers could change. This is very useful if you want to modify the assembly language output.

-s uses the small code model. (See Section 3.3.3 on the next page.)

-v produces code that can run in a multiprocess environment, where all variables may be considered volatile. Use this option when you compile modules that access variables which may be modified by another task (process). In general, code generated this way is significantly less efficient.

-x checks for overflow conditions of the runtime stack. The C compiler uses two stacks that grow together; unless you use the -x option, there is no automatic checking for stack overflow at run time. (See Section 3.3.4 on the next page.)

-z retains 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 do not specify the -z option, the code generator deletes the input (intermediate) file.

3.3.2 Pointers to Named Variables (-a Option)

You don't have to use -a if `ptr` doesn't point to a named variable. For example, it is not necessary to use -a if `ptr` points to an element of a dynamically allocated or statically allocated array. Note that structures are not considered to be named variables.

When you don't use the -a option, the compiler:

- Remembers that a register contains a constant or the value of a named variable, so it does not regenerate code to load that value into a register, **and**
- Assumes that an assignment of the form `*ptr = ...` does not assign a value to a named variable.

Under normal circumstances, the compiler cannot know which named variable an assignment will affect. Thus, when the compiler encounters such an assignment, it must forget the contents of all the registers that it assumed contained the values of named variables. When you use the -a option, the compiler generates less efficient code because it forgets these registers' contents and has to regenerate the code; thus, you should use this option sparingly.

3.3.3 Small Code Model (-s Option)

The compiler normally generates CALLA instructions; if you use -s, the compiler generates CALLR instructions. CALLR instructions are shorter and faster than CALLA instructions, but they limit the CALL range to ±32K words of the current PC.

Be sure that if you use the small code model, you don't generate calls outside of the 32K-word range; otherwise, your code won't run. You can verify that you conform to this limit by checking the link map (32K words translates to 0x80000 in bit addresses).

You can mix small-model code with other code as long as the small-model code conforms to the CALL restrictions. For example, if a module contains a group of functions that only call each other, and the size of this compiled module is 32K words or less, you can compile it with the -s option. You can then link this module with modules that weren't compiled with -s, as long as the small-code module doesn't call any code that is more than 32K words away. Small-model code *can* call functions outside the small-code module, as long as the called function is within the 32K-word limit.

3.3.4 Checking for Stack Overflow (-x option)

When you use -x, the code generator checks for stack overflow at the beginning of each function (after allocating the local frame). It does this by calling another function, s$check, that compares the two stack pointers.

- If the stacks don't overlap, s$check simply returns.
- If the stacks collide, s$check takes a TRAP 29. You can modify s$check to perform some other type of action; the source module for s$check is `scheck.asm`, which is a member of `rts.src`.

Note that there is usually no way to recover from a stack overflow. If the stack overflows, you can't use it, and thus you can't use C code for the abort process.

3.4 Compiling and Assembling a Program

The compiler creates a single assembly language source file as output, which you can assemble and link to form an executable object module. You can compile several C source programs, assemble each of them, and then link them together. (The *TMS34010 Assembly Language Tools User's Guide* describes the TMS34010 assembler and linker.)

Example 3-1 and Example 3-2 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 3-1 shows how you can accomplish this by invoking the preprocessor, the parser, the code generator, and the assembler in separate steps. Example 3-2 shows how you can use a batch file for compiling and assembling a file in one step.

Example 3-1. Method 1 - Invoking Each Tool Individually

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

```
gspcpp program
C Pre-Processor,        Version 3.xx
(c) Copyright 1988, Texas Instruments Incorporated
```

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

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

```
gspcc program
C Compiler,             Version 3.xx
(c) Copyright 1988, Texas Instruments Incorporated
   "program.c"  ==> main
```

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

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

```
gspcg program
C Codegen,              Version 3.xx
(c) Copyright 1988, Texas Instruments Incorporated
   "program.c"  ==> main
```

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

4) Assemble `program.asm`:

```
gspa program
COFF Assembler,         Version 3.xx
(c) Copyright 1988, Texas Instruments Incorporated
 PASS 1
 PASS 2

No Errors, No Warnings
```

This creates an output file named `program.obj`

Two batch files, `gspc` and `gspq`, are included as part of the TMS34010 C compiler package. The batch files expect C source files as input; each produces object files that can be linked. The batch files are essentially the same; however, `gspc` produces diagnostic and progress messages while `gspq` is a

"quiet" batch file that produces no messages. In addition, gspq deletes the intermediate asm file. To invoke the batch files, enter:

gspc *input file*	***or***	**gspq** *input file*

gspc/gspq name the batch files that invoke the tools.

input file names a C source file. If you don't specify a filename, the batch files will prompt you for one. Each batch file expects the input file to have an extension of *.c*. ***Do not specify an extension for the input file***; doing so may harm the input file.

The batch files only accept filenames as input; you cannot pass command options to the batch file. (If you want to use options, you must modify the batch files.) The batch files use the input filename to create and name the intermediate files and the output object file. The output file has the same name as the input filename, except the output file has an extension of ***.obj***. You can specify multiple input files to the batch file; for example,

```
gspc  file1 file2 file ...
```

Example 3-2 uses the gspc batch file to compile and assemble a C source file named program.c. (You could also use gspq, but it would not produce the messages shown in Example 3-2.)

Example 3-2. Method 2 - Using the Batch File

```
gspc program
---[program]---
C Pre-Processor,      Version 3.xx
(c) Copyright 1988 Texas Instruments Incorporated
C Compiler,           Version 3.xx
(c) Copyright 1988 Texas Instruments Incorporated
   "program.c"  ==> main
C Codegen,            Version 3.xx
(c) Copyright 1988 Texas Instruments Incorporated
   "program.c"  ==> main
COFF Assembler,       Version 3.xx
(c) Copyright 1988 Texas Instruments Incorporated
 PASS 1
 PASS 2

No Errors, No Warnings
       Successful Compile of Module  program
```

Note that the batch files do not create listing files. If you used gspc, you can create a listing file by invoking the assembler again with the -l option (lowercase L) and using *filename*.asm as the input file. For example,

```
gspc  program
gspa program -l
```

(You can't do this if you use gspq, because gspq deletes the .asm file.) If you want to create a listing file each time you use gspc, modify the batch file so that it invokes the assembler with the -l option.

3.5 Linking a C Program

The TMS34010 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:

gsplnk -c *filenames -o name.out* **-l rts.lib [-l flib.lib]**
or
gsplnk -cr *filenames -o name.out* **-l rts.lib [-l flib.lib]**

gsplnk is the command that invokes the linker.

-c/-cr are options that tell the linker to use special conventions that are defined by the C environment.

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

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

rts.lib/ flib.lib `rts.lib` is an archive library that contains C runtime-support functions, and `flib.lib` is the floating-point arithmetic library. (The -l option tells the linker that a file is an object library.) Both libraries are shipped with the C compiler. If you're linking C code, you must use `rts.lib`; you only need `flib.lib` if you're using the floating-point functions. Whenever you specify a library as linker input, the linker includes and links only those library members that resolve undefined references.

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

```
gsplnk -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 *TMS34010 Assembly Language Tools User's Guide.*

3.5.1 Runtime Initialization and Runtime Support

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

```
gsplnk  -c  -l rts.lib ...
```

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

When a C program begins running, it must execute `boot.obj` first. The symbol `_c_int00` is the starting point in `boot.obj`; if you use the -c or -cr option, then `_c_int00` is automatically defined as the entry point for the program. If your program begins running from reset, you should set up the reset vector to generate a branch to `_c_int00` so that the TMS34010 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.
- Calls exit when main returns.

Section 6 describes additional runtime-support functions that are included in rts.lib. If your program uses any of these functions, you must link rts.lib with your object files.

3.5.2 Sample Linker Command File

Figure 3-5 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.

```
/*****************************************************/
/*          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                */
```

Figure 3-5. An Example of a Linker Command File

- The command file 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 model of autoinitialization.

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

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

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

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

- Finally, the command file lists all the object libraries that the linker must search. (The libraries are specified with the -l linker option.) Since 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 support 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 in.

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

```
gsplnk  link.cmd
```

This example uses the default memory allocation described in Section 9 of the *TMS34010 Assembly Language Tools User's Guide*. If you want to specify different MEMORY and SECTIONS definitions, refer to that user's guide.

3.5.3 Autoinitialization (ROM and RAM Models)

The C compiler produces tables of data for autoinitializing global variables. (Section 5.8.2.1, page 5-25, 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 linker option)

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

  For more information about the ROM model, see Section 5.8.2.2 on page 5-26.

- **RAM Model** (-cr linker option)

  Global variables are initialized at *load time*. A loader copies the initialization routine into the variables in the .bss section; thus, no runtime initialization is performed at boot time. This enhances performance by reducing boot time and saving memory used by the initialization tables.

  For more information about the RAM model, see Section 5.8.2.3 on page 5-27.

3.5.4 The -c and -cr Linker Options

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

- The symbol `_c_int00` is defined as the program entry point; it identifies the beginning of the C boot routine in `boot.obj`. When you use -c or -cr, `_c_int00` is automatically referenced; this ensures that `boot.obj` is automatically linked in from the runtime-support library `rts.lib`.

- The .cinit output section is padded with a termination record so that the boot routine (ROM model) or the loader (RAM model) can identify the end of the initialization tables.

- In the ROM model (-c option), the linker defines the symbol `cinit` as the starting address of the .cinit section. The C boot routine uses this symbol as the starting point for autoinitialization.
- In the RAM model (-cr option):
  - The linker sets the symbol `cinit` to -1. This indicates that the initialization tables are not in memory, so no initialization is performed at run time.
  - The STYP_COPY flag (010h) is set in the .cinit section header. STYP_COPY is the special attribute that tells the loader to perform autoinitialization directly and not to load the .cinit section into memory. The linker does not allocate space in memory for the .cinit section.

3.6 Archiving a C Program

An archive file (or library) is a partitioned file that contains complete files as members. The TMS34010 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 *TMS34010 Assembly Language Tools User's Guide* contains complete instructions for using the archiver.

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

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

For more information about the archiver, see the *TMS34010 Assembly Language Tools User's Guide*.

Section 4

The TMS34010 C Language

The C language that the TMS34010 C compiler supports is based on the Unix[6] System V C language that is described by Kernighan and Ritchie, with several additions and enhancements to provide compatibility with ANSI C. The most significant differences are:

- The addition of *enum* data type.
- A member of a structure can have the same name as a member of another structure (unique names aren't required).
- Pointers to bit fields within structures are allowed.
- Structures and unions may be passed as parameters to functions, returned by functions, and assigned directly.

This section compares the C language compiled by the TMS34010 C compiler to the C language described by Kernighan and Ritchie. It presents only the *differences* in the two forms of the C language. The TMS34010 C compiler supports standard Kernighan and Ritchie C except as noted in this section.

Throughout this section, 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 section include:

Section		Page
4.1	Identifiers, Keywords, and Constants	4-2
4.2	TMS34010 C Data Types	4-4
4.3	Object Alignment	4-6
4.4	Conversions	4-6
4.5	Expressions	4-7
4.6	Declarations	4-8
4.7	Initialization of Static and Global Variables	4-10
4.8	asm Statement	4-10
4.9	Lexical Scope Rules	4-11

6 UNIX is a registered trademark of AT&T.

4.1 Identifiers, Keywords, and Constants

K&R 2.2 - Identifiers

- In TMS34010 C, the **first 31 characters of an identifier are significant** (in K&R C, 8 characters are significant). This also applies to external names.
- **Case is significant**; uppercase characters are different from lowercase characters in all TMS34010 tools. This also applies to external names.

K&R 2.3 - Keywords

TMS34010 C reserves **three additional keywords**:

```
asm
void
enum
```

K&R 2.4.1 - Integer Constants

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

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

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

K&R 2.4.3 - Character Constants

In addition to the escape codes listed in K&R, the TMS34010 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

TMS34010 C supports both single-precision and double-precision floating-point constants. You can append a letter suffix to a floating-point constant to specify its type.

- A floating-point constant that is used in an expression is normally treated as a double-precision constant (type double). If you want to use a single-precision constant, use *F* as a suffix to identify the constant as type float.
- ANSI standard C supports a long double type that can provide more precision than a double. Long doubles are specified with an *L* suffix (like long ints). TMS34010 C does not directly support long doubles;

it treats them as ordinary doubles. The *L* suffix is supported to provide compatibility with ANSI C.

Examples of floating-point constants include:

```
1.234;              /* double */
1.0e14F;            /* float  */
3.14159L;           /* double */
```

Suffixes can be upper or lower case.

Added type - Enumeration Constants

An enumeration constant is **an additional type of integer constant** that is not described by K&R. An identifier that is declared as an enumerator can be used in the same manner as an integer constant. (For more information about enumerators, see Section 4.6 on page 4-8.)

K&R 2.5 - String Constants

- K&R C does not limit the length of string constants; *however,* TMS34010 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 to autoinitialize arrays of characters.

4.2 TMS34010 C Data Types

K&R 4.0 - Equivalent Types

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

K&R 4.0 - Additional Types

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

```
void  *p, *malloc();
char  *c;
int   i;

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

K&R 4.0 - Derived Types

TMS34010 C allows any type declaration to have **up to six derived types.** Constructions such as *pointer to*, *array of*, and *function returning* can be combined and applied a maximum of six times.

For example:

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

translates as:

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

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

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

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

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

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

K&R 2.6 - Summary of TMS34010 Data Types

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

4.3 Object Alignment

- All objects except structure members and array members are aligned on a 16-bit (one-word) boundary. In other words, with the exception of structure and array members, all objects begin at bit addresses whose four LSBs are zeros. In addition, because of the TMS34010's bit addressability, a pointer can point to any bit address. Signed objects of less than 16 bits are sign-extended to 16 bits. Unsigned objects of less than 16 bits are zero-extended to 16 bits.

- Structure or array members are not aligned to 16-bit boundaries. However, the structure or array itself begins at a 16-bit boundary. In the case of an array of structures, only the first structure in the array is constrained to begin on a 16-bit boundary.

For additional information on array alignment, see Packing Structures and Manipulating Fields, Packing Structures and Manipulating Fields, on page 5-5.

4.4 Conversions

K&R 6.1

Integer objects are always widened to 32 bits when passed as arguments to a function. Signed objects of less than 32 bits are sign-extended to 32 bits; unsigned objects of less than 32 bits are zero-extended to 32 bits.

The type *char* is signed and is therefore sign-extended when widened to *integer* type. Sign extension can be disabled by using the type *unsigned char*.

K&R 6.3

- Types *float* and *double* are converted to type *integer* by truncation.

- In K&R C, all floating-point arithmetic is performed on double-precision values. In ANSI and TMS34010 C, however, single-precision values can be used for calculating any expression in which both operands have type float. If either operand in an expression has type double, the other operand is converted to a double and the arithmetic is performed on double-precision values. Floating-point constants have type double **unless** they have an *F* suffix; also, integers have type float when they are implicitly converted. Single-precision arithmetic is significantly faster, but causes a loss of precision. The following examples illustrate cases in which a single-precision value is used as-is or is converted to a double:

```
float  f;
double d;
int    i;

f + f;          /* Single Precision */
f + d;          /* Double Precision */
f + 1;          /* Single Precision */
f + 1.0;        /* Double Precision */
f + 1.0F;       /* Single Precision */
d * (f + i);    /* Add using Single, multiply */
                /* using Double     */
```

K&R 14.4

Pointers and integers (or longs) may be freely converted, since each occupies 32 bits of storage. Pointers to one data type can also be converted to pointers to another data type, since the TMS34010 has no alignment restrictions and all pointers are the same size.

4.5 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 7.1 - Primary Expressions

In TMS34010 C, functions can return structures or unions.

The restriction of three array dimensions does not apply to expressions, because [ ] is treated as an operator.

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 eight. (Eight is the number of bits in a character.) *Sizeof* 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.

4.6 Declarations

K&R 8.1 - Register Variables

- A function can have a maximum of eight register variables; the limit applies to the combination of register arguments and local register variables.
- Any scalar type variable that is less than 32 bits (such as int, float, or pointer) can be declared as a register. Other types (such as struct, double, or arrays) cannot be declared as registers.
- A *register* declaration of an invalid type or a declaration after the first eight registers have been declared is treated as a normal *auto* declaration.
- Function arguments can be declared as type register. Such arguments are passed on the stack in the normal way; the function pops them off into registers and they are treated like normal register variables for the duration of the function.

K&R 8.2 - Type Specifiers in Declarations

- In addition to the type-specifiers listed in K&R, objects may be declared with *enum* specifiers.
- TMS34010 C allows **more type name combinations** than K&R C. The adjectives *long* and *short* can be used with or without the word *int*; the meaning is the same in either case. The word *unsigned* can be used in conjunction with any *integer* type or alone; if alone, *int* is implied. *long float* is a synonym for *double*. Otherwise, only one type specifier is allowed in a declaration.

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

- Contrary to K&R, TMS34010 C allows functions to return structures and unions.
- Structures and unions can be used as function parameters and can be directly assigned.

K&R 10 - External Definitions

Formal parameters to a function may be declared as type *struct, union,* or *enum* (in addition to the normal function declarations), since TMS34010 C allows you to pass such objects to functions.

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

- Since the TMS34010 is bit-addressable, structure members are not aligned in any way. The statement in K&R about alignment or boundaries for structure members *does not apply to TMS34010 C*. Any field with width zero (normally used to force alignment) is ignored. However, bit fields are limited to a width of 32 bits.
- Any integer type may be declared as a field. Fields are treated as signed unless declared otherwise. Also, contrary to K&R, pointers to fields are legal in TMS34010 C.

- K&R states that structure and union member names must be mutually distinct. In TMS34010 C, **members of different structures or unions can have the same name.** However, this requires that references to the member be fully qualified through all levels of nesting.

- TMS34010 C **allows** assignment to and from structures, passing structures as parameters, and returning structures from functions.

- K&R contains a statement about the compiler determining the type of a structure reference by the member name. Since TMS34010 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 TMS34010 C. The syntax of an enumeration declaration is similar to that of a structure or union. The keyword *enum* is substituted for *struct* or *union*, and a list of enumerators is substituted for the list of members.

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

The list of enumerators is simply a comma-separated list of identifiers. Each identifier can be followed by an equal sign and an integer constant. If there is no enumerator 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; the next enumerator is assigned that value + 1, the next is the preceding value + 1, etc. The assigned value may be negative, but the increments are still by positive 1.

The size of an object of type *enum* is determined as follows: if any of the object's enumerators have negative values, the object occupies 32 bits. Otherwise, the object occupies the minimum number of bits required to represent the largest enumerator value and is considered to be unsigned.

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; thus, they can be used in arithmetic expressions, case expressions, etc. In addition, explicit integer expressions may 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
```

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

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

4.7 Initialization of Static and Global Variables

K&R 8.6

An important difference between K&R C and TMS34010 C is that **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, you can use the linker to accomplish this. In the linker control file, use a fill value of 0 in the .bss section:

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

4.8 asm Statement

Additional Statement

TMS34010 C supports another statement that is 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(*"assembly language statement"***);**

The assembly language statement must be enclosed in double quotes. All the usual character string escape codes retain their definitions. The assembler statement is copied directly to the assembler source file. Note that the assembly language 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 sequential statements 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.

Warning:

Be extremely careful not to disrupt the C environment with asm commands. The compiler does not check the inserted instructions. Inserting jumps and labels into C code can cause unpredictable results in variables manipulated in or around the inserted code. The asm command is provided so you can access features of the hardware, which by definition C is unable to access. Specifically, do not use this command to change the value of a C variable; however, you can use it safely to read the current value of a variable.

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

The *asm* command is very useful in the context of register variables. A register variable is a variable in a C program that is declared by the user to reside in a machine register. TMS34010 C allows up to eight machine registers to be allocated to register variables. These eight registers, combined with the *asm* command, provide a means of manipulating data independently of the C environment.

4.9 Lexical Scope Rules

K&R 11.1

The lexical scope rules in K&R also apply to TMS34010 C, 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.

Section 5

Runtime Environment

This section describes the TMS34010 C runtime environment. To ensure successful execution of C programs, it is critical that all runtime code maintain this environment. If you write assembly language functions that interface to C code, follow the guidelines in this section.

Topics in this section include:

Section		Page
5.1	Memory Model	5-2
5.2	Register Conventions	5-6
5.3	Function Structure and Calling Conventions	5-8
5.4	Interfacing C with Assembly Language	5-12
5.5	Interrupt Handling	5-16
5.6	Integer Expression Analysis	5-17
5.7	Floating-Point Support	5-17
5.8	System Initialization	5-22

5.1 Memory Model

TMS34010 C treats memory as a single linear block that is partitioned into subblocks of code and data. Each block of memory generated by a C program will be 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 that are available, about any locations that are not available (holes), or about any locations that are reserved for I/O or control purposes. The compiler produces relocatable code, which allows the linker to allocate code and data into the appropriate memory spaces. Each block of code and data could be allocated individually into memory, but this is not a general practice (an exception to this is memory-mapped I/O, although physical memory locations can be accessed with C pointer types).

5.1.1 Sections

The compiler produces 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 (Introduction to Common Object File Format) of the *TMS34010 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 C boot routines 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 and the program stack.
  - It reserves space for use by the dynamic memory functions (malloc, calloc, and realloc).

Note that the *assembler* creates an additional section called .data; 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 four output sections. The complete program is made up of the compiler's three output sections plus the assembler's .data section. You can place these output sections anywhere in the address space, as needed, to meet system

requirements. The .text, .cinit, and .data sections are usually linked into either ROM or RAM. The .bss section must be linked into some type of RAM.

For more information about allocating sections into memory, see Section 9 (the Linker Description) of the *TMS34010 Assembly Language Tools User's Guide.*

5.1.2 Stack Management

The C compiler uses two stacks; Figure 5-1 illustrates these stacks in memory.

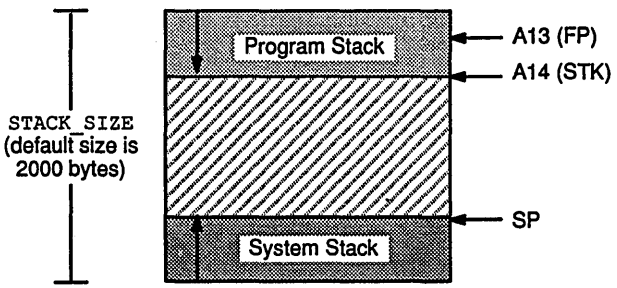

Figure 5-1. The Program and System Stacks

- The **program stack** is used for passing parameters to a function and for allocating the local frame for a function.
- The **system stack** is used for saving the status of the calling function (that is, for saving the values in registers) and the return address.

The C initialization routine, `boot.c`, allocates memory for both stacks in the .bss section. This memory is allocated as a single, static array named `SYS_STACK`. The boot routine defines a constant named `STACK_SIZE` that determines the size of `SYS_STACK`. The default stack size is 2000 bytes. You can change the amount of memory that is reserved for the stack by following these steps:

1) Extract `boot.c` from the source library `rts.src`.

2) Edit `boot.c`; change the value of the constant `STACK_SIZE` to the desired stack size.

3) Recompile `boot.c` and replace the resulting object file, `boot.obj`, in the object library `rts.lib`

4) Replace the copy of `boot.c` that's in `rts.src` with the edited version.

The two stacks grow toward each other. The program stack grows from the bottom of the array (the lowest address) to higher addresses; the system stack grows from the top of the array (the highest address) down to lower addresses. *Do not modify the way the stacks grow!*

The compiler uses three registers to manage the stack:

SP is the stack pointer for the **system stack.**
A14 (STK) is the stack pointer for the **program stack.**

A13 (FP) is the **frame pointer**; it points to the beginning of the current local frame. (The local frame is an area on the program stack used for storing arguments and local variables.)

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

5.1.3 Dynamic Memory Allocation

The runtime-support library supplied with the compiler contains several functions (such as malloc, calloc, and realloc) that allow you to dynamically allocate memory for variables at 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 `MEMORY_SIZE` that determines the size of the memory pool; the default size is 1000 bytes. 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) Replace the copy of `memory.c` that's in `rts.src` with the edited version.

5.1.4 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 about autoinitialization, refer to Section 5.8.2 on page 5-23.

5.1.5 Allocating Memory for Static and Global Variables

A unique, contiguous space is allocated in the .bss section for each external or static variable that is declared in a C program. The linker determines the address of the global variables when it allocates the .bss section. The compiler ensures that space for these variables is allocated in multiples of words, so that each variable is aligned on a word boundary. You should allocate .bss into RAM when you link the program.

5.1.6 Packing Structures and Manipulating Fields

When the compiler allocates space for a structure, it allocates the exact amount of memory needed to contain all of the structure's members. Fields are allocated as many bits as requested; enumerated types are allocated as few bits as possible to hold the maximum value of that type; bytes are allocated eight bits, and so on.

The C compiler follows standard C practice for mapping structures, with one exception: a field of width zero does not force word alignment. Because of the TMS34010's bit-addressability, word alignment in a structure does not necessarily produce more efficient code. However, a field that straddles word boundaries does take longer to access, since the TMS34010 must fetch more than one word. You should be very careful when you're defining structures or arrays of structures; try to avoid defining fields that cross word boundaries.

If a structure is declared as an external or static variable, it is always placed on a word boundary and is allocated space rounded up to a word boundary. However, when an array of structures is declared, no rounding of size is used; exactly enough space is allocated to hold each structure element in contiguous bits of memory.

5.1.7 Array Alignment

In ANSI standard C, as well as K&R C, arrays are expected to always align their elements on a word boundary, with the exception of bytes, which may be aligned on a byte boundary. The TMS34010's bit-addressability makes this restriction both unimportant and inefficient; so, in TMS34010 C, arrays have no internal alignment. Each array element is allocated exactly as much space as needed, with no space between adjacent elements.

> **Note:**
>
> Like structures, a carefully defined array (with no elements overlapping word boundaries) will allow the program to run faster. Pixel arrays are usually aligned in this manner.

If an array is declared as an external or static variable, the first element of the array is placed on a word boundary and the array is allocated space rounded up to a word boundary.

This method of handling an array allows more control over the environment than standard C allows. Bit arrays and pixel arrays are directly accessible (a necessity for a graphics environment), and memory-mapped I/O is more straightforward.

5.2 Register Conventions

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

5.2.1 Dedicated Registers

The C environment reserves three registers. **Do not** modify these registers in any other manner than that described in Section 5.3, Function Structure and Calling Conventions, page 5-8.

SP points to the top of the system stack.
A14 (STK) points to the program stack.
A13 (FP) points to the beginning of the currently active frame.

In addition, the C compiler assumes certain information about bits in the status register. Specifically, it assumes that FS1 (field size 1) is 32 within a C function. FS0, however, can be changed in a function without being restored.

5.2.2 Using Registers

A function can usually use registers A0 through A12, however:

- When a function is called, it must save the contents of each register that it uses; it must restore these registers before it returns to the caller. Register A8 is the only exception; its contents do not have to be saved or restored.
- If a function returns an integer value or a pointer, the value must be placed in A8.

The code generator uses the A-file registers for the following purposes:

Expression analysis	A0 through A11
Return value/Scratch	A8
User register variables	A9, A10, A11, A12, A0, A2, A4, A6

The C compiler doesn't use registers B0 through B14.

Expression-analysis registers are allocated from high to low registers, based on availability and current use. (All integer expression analysis uses 32-bit math.)

> **Note:**
>
> The compiler constantly tracks the contents of registers and attempts to reuse register data whenever possible. Therefore, it is inadvisable to use inline assembly language or any other method to modify a register that a function is using. Use the -r code generator option to produce information about register use.

5.2.3 Register Variables

The C compiler uses up to eight register variables within a function. The compiler allocates the first four variables from registers A9 through A12 in ascending order; the other variables are allocated from other available registers. If more than eight register variables are declared, the excess are treated as normal variables. A register variable can contain any integer type, a pointer to any type, or a float (doubles or structures are not allowed); however, register variables of type short and char are treated as long integers.

Using register variables can greatly increase code efficiency for some statements (in some cases, the code may be twice as efficient). Since the compiler does not track operations involving register variables, you can manipulate them as desired (even with asm statements).

5.3 Function Structure and Calling Conventions

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

Figure 5-2 illustrates a typical function call. Parameters are passed to this function, the function uses local variables, but no value is returned.

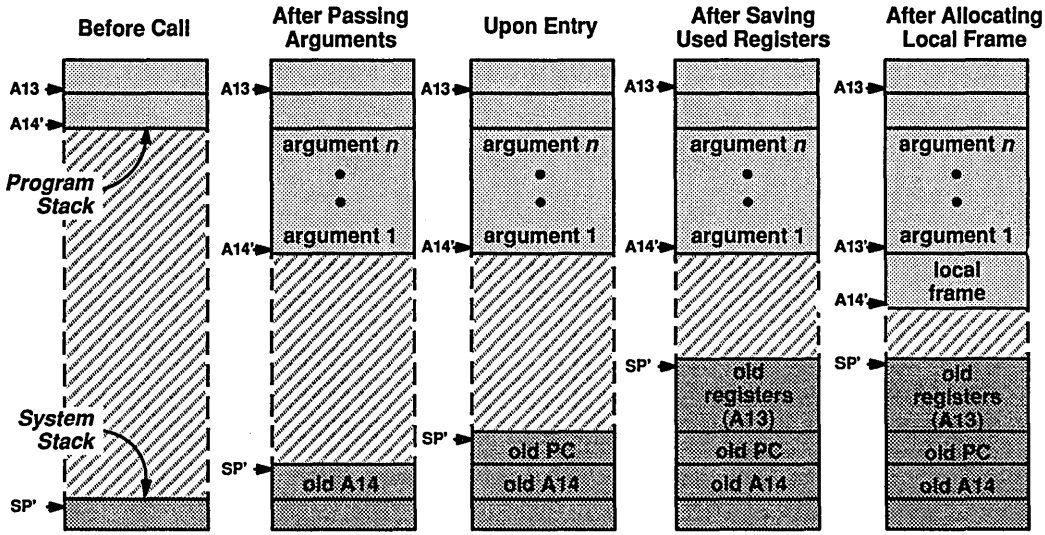

Figure 5-2. An Example of a Function Call

5.3.1 Responsibilities of a Calling Function

A function performs the following tasks when it calls another function. The steps below show examples of the code that the compiler might generate for a particular step. For these code examples, assume that a C function calls another function `f` that has three arguments; the function call is:

```
f(arg1, arg2, arg3);
```

1) If the called function returns a double or a float, the caller allocates space on the program stack for the return value. The caller must allocate this space even if it doesn't use the return value.

2) It saves the program stack pointer (A14) on the system stack. The caller generates the following:

```
MOVE  STK, -*SP, 1
```

(This is only done when the caller passes arguments or when the called function returns a float or a double – that is, when the program stack is affected.)

3) It pushes the arguments on the program stack in reverse order (pushes the rightmost declared argument first and pushes the leftmost declared argument last). If the called function does not have any arguments, the caller **must not** push any. If the called function expects one or more arguments, the caller must push **at least one** argument. The caller generates the following code when it pushes the arguments:

```
MOVE  @_arg3, *STK+, 1
MOVE  @_arg2, *STK+, 1
MOVE  @_arg1, *STK+, 1
```

All integer types that are passed as arguments are widened to 32-bit integers. All floating-point arguments are converted to double-precision values. Structures are rounded up to the next word boundary.

4) If the called function returns a structure, the caller pushes the address of the structure onto the program stack. (For more information about functions that return structures, see Section 5.3.5.)

5) It calls the function by generating the following instruction:

```
CALLA _f
```

Note that within a called function, FS1 must equal 32.

The called function restores the program stack pointer (effectively popping the arguments), so there is no need for the calling function to perform any cleanup after the function call.

5.3.2 Responsibilities of a Called Function

A called function must perform the following tasks. The steps below show examples of the code that the compiler might generate for a particular step.

1) If the function modifies any registers, it saves them on the system stack. If it uses the FP, it must save it with the other registers. If the called function must save multiple registers on the system stack, it can use the MMTM instruction as shown below:

```
MMTM  SP, A5, A7, FP
```

It is not necessary to save register A8 or the status register. Note that the C compiler doesn't use registers B0–B14 so that assembly-language functions can use them.

2) It executes the code for the function.

3) If the function returns an integer or a pointer, it returns the value in register A8; for example,

```
MOVE  @result, A8, 1
```

If the function returns a double or a float, then the caller has allocated space on the program stack for the return value; the called function copies the value into that space. If the function returns a structure, see Section 5.3.5.

4) It restores the caller's environment.

   a) If the function has arguments or returns a float or a double, it must restore the caller's stack. To do this, it moves the value out of the system stack to the program stack pointer (register A14). The STK is stored on the system stack below the saved registers and the old

PC. STK is accessed as *SP(*offset*), where the *offset* = [number of saved registers + 1] times 32. If the calling function saved three registers, it would restore STK with the following instruction:

```
MOVE  *SP(128), STK, 1
```

b) It restores the saved registers. If local variables were allocated, it must also restore the FP along with the other registers. If the called function must restore multiple registers, it can use the MMFM instruction as shown below:

```
MMFM  SP, A5, A7, FP
```

It is not necessary to restore the status register; however, if FS1 has been changed, it must be restored to the value 32 and FE0 must equal 0.

5) It executes an RETS instruction. If the function has arguments or returns a value on the stack, it executes an `RETS 2` instruction; this pops both the return address and the caller's old STK off the stack. If the function has no arguments and doesn't return a float or a double, it can execute an `RETS 0` instruction.

5.3.3 Setting up the Local Frame

In addition to the actions listed in Section 5.3.2, a called C function that has arguments or local variables must perform the following actions to setup the local frame. These additional steps are performed immediately following step 1 above.

1) It sets the new frame pointer to the current program stack pointer (A14):

```
MOVE  STK, FP
```

2) It allocates the frame by adding its size to the program stack pointer:

```
ADDI  128, STK
```

If the called function has no local variables or arguments, then it has no need for local temporary storage and these steps are not necessary.

5.3.4 Accessing Arguments and Local Variables

A function accesses its arguments and local variables indirectly through the FP (A13). The FP always points to the bottom of the local frame (where the first local variable is). The first local variable is accessed as `*FP(0)`. Other local variables are addressed with increasing offsets, up to a maximum of 32,768. Arguments are accessed similarly, but with negative offsets from the FP (up to a maximum of -32,767). The first argument is accessed as `*FP(-32)`.

> **Note:**
>
> All integer arguments are widened to 32-bit integers. All floating-point arguments are converted to doubles. All structures that are passed as arguments are rounded up to the next word boundary.

5.3.5 Returning Structures from Functions

A special convention applies to functions that return structures. The caller allocates space for the structure, and passes the address of the return space by pushing the address on the stack just before calling the function. To return a structure, the called function then copies the structure to the memory block that the address points to.

This methods allows the caller to be smart about telling the called function where to return the structure. For example, consider the following statement:

```
s = f();
```

where `s` is a structure and `f` is a function that returns a structure. The caller can simply push the address of `s` onto the stack and call `f`. Function `f` then copies the return structure directly into `s`, performing the assignment automatically.

If the caller does not use the return value, then it pushes a value of 0 onto the stack instead of pushing an address. This tells the function not to copy the return structure.

Be careful to properly declare functions that return structures, both when they are called (so the caller pushes the address correctly) and when they are defined (so the function knows where to copy the result).

5.4 Interfacing C with Assembly Language

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

- Use separate modules of assembled code and link them with compiled C modules (see Section 5.4.1). This is the most versatile method.
- Use inline assembly language, imbedded directly in the C source (see Section 5.4.2).
- Modify the assembly language code that the compiler produces (see Section 5.4.3).

5.4.1 Assembly Language Modules

Interfacing with assembly language functions is straightforward if you follow the calling conventions defined in Section 5.3 and the register conventions defined in Section 5.2. C code can access variables and call functions that are defined in assembly language, and assembly code can access C variables and call C functions.

Follow these guidelines to interface assembly language and C:

1) All functions, whether they are written in C or assembly language, must follow the conventions outlined in Section 5.3 (page 5-8).

2) You must preserve any registers that are modified by a function, except register A8. When returning from a function, FS1 must equal 32.

3) Interrupt routines must save **all** the registers they use. (For more information about interrupt handling, see Section 5.5, page 5-16.)

4) If the caller passes arguments or if the called function returns a float or a double, save the program stack pointer (A14) by pushing it on the system stack, then push any arguments on the program stack in reverse order.

5) Functions must return values correctly according to their C declarations. Integers and pointers are returned in register A8. All floating-point values are returned on the stack. Section 5.3.5 discusses returning structures.

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

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

8) Any object or function declared by an assembly language routine that is to be accessed or called from C must be declared with the .global as-

sembler 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 must resolve.

5.4.1.1 An Example of an Assembly Language Function

Example 5-1 illustrates a C function called `main` which calls an assembly language function called `asm_func`. The `asm_func` function has one argument which is a pointer to an integer. `asm_func` calls another C function called `c_func` with one argument which is a global variable named `gvar`. `asm_func` takes the value returned from `c_func` and stores it in the integer pointed to by its single argument.

Example 5-1. An Assembly Language Function

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

main()
{
   int i, j;
   i = asm_func(&i);
}
```

(a) C Program

```
FP      .set    A13             ; frame pointer
STK     .set    A14             ; program stack pointer
        .global _gvar           ; declare global variable
        .global _c_func         ; declare C function
        .global _asm_func       ; declare this function

_asm_func:
        MMTM    SP,A7,FP        ; save registers on SP
        MOVE    STK,FP          ; set up FP
        MOVE    *FP(-32),A7,1   ; get argument
        MOVE    STK,-*SP,1      ; function call: save STK
        MOVE    @_gvar,*STK+,1  ; push argument
        CALLA   _c_func         ; call function
        MOVE    A8,*A7,1        ; result in A8

        MOVE    *SP(96),STK,1 ; restore caller's STK
                              ; (pop arguments)
        MMFM    SP,A7,FP      ; restore saved registers
        RETS    2             ; return & pop caller's STK
```

(b) Assembly Language Program

In the C program in Example 5-1, the *extern* declaration of `asmfunc` is optional, since the function returns an int. Like C functions, assembly functions need only be declared if they return non-integers.

In the assembly language code in Example 5-1, note the underscores on all the C symbol names when used in the assembly code.

5.4.1.2 Defining Variables in Assembly Language

It is sometimes useful for a C program to access variables that are defined in assembly language. There are several methods for defining a variable in assembly language; accessing the variable is straightforward:

In Assembly Language:
1) Define the variable:
   a) Use the .bss directive to define the variable in the .bss section.
   or b) Define the variable in a named, initialized section (.sect).
   or c) Define the variable in a named, uninitialized section (.usect).
2) Use the .global directive to make the definition external.
3) Remember to precede the variable name with an underscore.

In C:
4) Declare the variable as *extern*, and access it normally.

The C compiler uses the first method by defining variables in the .bss section.

Example 5-2 *a* shows examples that use these three methods to define variables. Example 5-2 *b* shows how you can use C code to access the first variable defined in Example 5-2 *a*; you can access the other variables similarly.

Example 5-2. Accessing an Assembly-Language Variable from C

```
**  Method 1:
**  Define variable var in the .bss section
**
        .bss     _var, 32       ; Define the variable
        .global  _var           ; Declare it as external

**  Method 2:
**  Define variable table in a named, initialized section
**
        .sect    "more_vars"
_table  .word    01234h         ; Define the variable
        .word    05678h
        .word    09ABCh
        .global  _table         ; Declare it as external

**  Method 2:
**  Define variable buffer in a named, uninitialized
**  section
**
_buffer .usect   "ramvars", 32 ; Declare the variable
        .global  _buffer        ; Declare it as external
```

(a) Assembly Language Program

```
/* This examples shows you can access the variable */
/* named var, which was defined above; you can     */
/* access the other variables similarly.           */

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

(b) C Program

You can use a named section to define as many variables as you like (in the same way that the compiler uses .bss for multiple variables). It is not necessary to use a .sect or .usect section for each new variable unless you want to allocate it in memory separately from other variables. For example, you may want to define a lookup table in its own named section if you don't want to allocate it into RAM with the .bss section.

5.4.2 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 C code. See Section 4.8, page 4-10, for more information about the asm statement.

> **Warning:**
>
> **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 the C code may produces unpredictable results by confusing the register-tracking algorithms that the code generator uses. The asm statement is provided so that you can access features of the hardware which would be otherwise inaccessible from C.**
>
> **Do not change the value of a C variable; however, you can safely read the current value of any variable.**
>
> **In addition, do not use the asm statement to insert assembler directives that would change the assembly environment.**

The asm statement is also useful for inserting comments in the compiler output; simply start the assembly language statement with an asterisk:

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

5.4.3 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 5.4.2 about disrupting the C environment also apply to modifying compiler output.

5.5 Interrupt Handling

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

The C compiler uses a special naming convention for interrupt functions; such functions have names with the following format:

`c_int`*nn*

where *nn* is a two-digit number between 00 and 99 (for example, `c_int01`). Following this convention assures that the compiler treats the function as an interrupt function. The name `c_int00` is reserved for the system reset interrupt. This special interrupt routine initializes the system and calls the function `main`; `c_int00` does not save any registers since it has no caller.

Interrupt routines for any interrupt except `c_int00` must save any register used (with the exception of SP and STK), including A8. In a normal function, it is not necessary to save A8; however, in the case of an interrupt, A8 **must be saved**. The compiler uses the RETI instruction to return from an interrupt; RETI restores the ST register of an interrupted function.

A C interrupt routine is like any other C function - it can have local variables and register variables, it can access global variables, and it can call other functions. However, an interrupt routine should be declared without any arguments and it should not be called directly.

Any interrupt function can be used to handle any interrupt or multiple interrupts. The compiler does not generate any code specific to the particular interrupt, with the exception of the system reset interrupt (c_int00), which must be used as system reset and cannot have any local variables (since it is assumed that at system reset the stack has not yet been allocated).

To associate an interrupt function with an interrupt, the address of the interrupt must be placed in the proper interrupt vector. You can accomplish this by using the assembler and the linker to create a simple table of addresses at the proper location.

5.6 Integer Expression Analysis

All integer expression analysis is performed in the A-file registers using the TMS34010's 32-bit math instructions. All multiplication and division operations are performed in odd registers; for this reason, only A1, A3, A5, and A7 are used for general-purpose expression registers.

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

If the compiler runs out of registers to use, it selects a used register and saves its contents on the local frame, temporarily freeing the register for reuse.

5.7 Floating-Point Support

The TMS34010 C compiler supports floating-point functions for both single-precision (32-bit) and double-precision (64-bit) values. All floating-point arguments are passed on the stack; floating-point return values are returned on the stack. Single-precision values are converted to doubles when they are passed to functions. Operations between two single-precision operands are performed in single-precision. Operations between a single-precision operand and a double-precision operand are performed in double-precision.

A custom package of floating-point routines is included with the C compiler; these functions do not follow standard C calling conventions. The calling conventions for these routines follow a classic operand stack:

- The compiler pushes the floating-point arguments onto the argument stack, then generates a call to a floating-point function.
- The floating-point function pops the arguments off the stack, performs the operation, and pushes the results back onto the stack.

Some floating-point functions expect integer arguments or return integer values; all integers are passed and returned in register A8.

Section 5.7.1 describes the floating-point formats used for these routines; Section 5.7.2 through Section 5.7.5 list the floating-point routines.

5.7.1 Floating-Point Formats

The compiler is unaware of the internal floating-point format; the only restriction the compiler places on a floating-point number is the representation of the number. This allows you to customize a floating-point package for your environment. Section 5.7.1.1 and Section 5.7.2 describe the floating-point format used by the floating-point routines that are included with the C compiler.

5.7.1.1 Single-Precision Floating-Point Format

Figure 5-3 illustrates the single-precision floating-point format. Single-precision values are represented in 32 bits: a sign bit (bit 31), an 8-bit biased exponent (bits 23–30), and a 23-bit mantissa (bits 0–22).

Figure 5-3. Single-Precision Format

Given a sign bit s, an exponent e, and a mantissa f, the value V of the floating-point number $X=(s,e,f)$ is:

- If $s=0$, $e=255$, and $f=0$, the $V = +\infty$
- If $s=1$, $e=255$, and $f=0$, the $V = -\infty$
- If $0<e<255$ and $f\neq 0$, then $V = (-1)^s \times 2^{e-127}(.f)$; V is not valid if $X=(s,e,f)$ is not a normalized floating-point number
- If $s=0$, $e=0$, and $f=0$, the $V = 0$
- For all other cases, V = not valid

Precision of single-precision values is greater than six decimal digits; the range includes:

- *Positive values:* 5.87747×10^{-39} to 1.70141×10^{38}
- *Negative values:* -1.70141×10^{38} to -5.87747×10^{-39}
- 0
- $\pm\infty$

5.7.1.2 Double-Precision Floating-Point Format

Figure 5-4 illustrates the double-precision floating-point format. Double-precision values are represented in 64 bits: a sign bit (bit 63) an 11-bit biased exponent (bits 52–62), and a 52-bit mantissa (bits 0–51).

Figure 5-4. Double-Precision Format

Given a sign bit s, an exponent e, and a mantissa f, the value V of the floating-point number $X=(s,e,f)$ is:

- If $s=0$, $e=2047$, and $f=0$, the $V = +\infty$
- If $s=1$, $e=2047$, and $f=0$, the $V = -\infty$

- If $0<e<2047$ and $f \neq 0$, then $V = (-1)^s \times 2^{e-1023}(.f)$; V is not valid if $X=(s,e,f)$ is not a normalized floating-point number
- If $s=0$, $e=0$, and $f=0$, the $V = 0$
- For all other cases, V = not valid

Precision of double-precision values is greater than 15 decimal digits; the range includes:

- *Positive values:* 1.11254×10^{-308} to 8.98847×10^{308}
- *Negative values:* -8.98847×10^{308} to $-1.11254 \times 10^{-308}$
- 0
- $\pm\infty$

5.7.2 Double-Precision Functions

Assume that `d1` and `d2` are double-precision floating-point values on the stack; `d1` is pushed first.

Function	Action
FD$ADD	Return `d1 + d2`
FD$SUB	Return `d1 - d2`
FD$SUB_R	Return `d2 - d1`
FD$MUL	Return `d1 * d2`
FD$DIV	Return `d1 / d2`
FD$DIV_R	Return `d2 / d1`
FD$INC	Return `d1 + 1.0`
FD$INCR	Return `d1 + 1.0` (*Note:* `d1` is not popped.)
FD$DEC	Return `d1 - 1.0`
FD$DECR	Return `d1 - 1.0` (*Note:* `d1` is not popped.)
FD$GE	Return `d1` $\geq$ `d2` (Return 1 or 0 in A8 and set status.)
FD$LE	Return `d1` $\leq$ `d2` (Return 1 or 0 in A8 and set status.)
FD$GT	Return `d1 > d2` (Return 1 or 0 in A8 and set status.)
FD$LT	Return `d1 < d2` (Return 1 or 0 in A8 and set status.)
FD$EQ	Return `d1 == d2` (Return 1 or 0 in A8 and set status.)
FD$NE	Return `d1 != d2` (Return 1 or 0 in A8 and set status.)
FD$NEG	Return `-d1` (*Note:* FD$NEG is also used for single-precision.)
FD$ZERO	Returns 1 if `d1 = 0`

5.7.3 Single-Precision Functions

Assume that `f1` and `f2` are single-precision floating-point values on the stack; `f1` is pushed first.

Function	Action
FS$ADD	Return `f1` + `f2`
FS$SUB	Return `f1` - `f2`
FS$SUB_R	Return `f2` - `f1`
FS$MUL	Return `f1` * `f2`
FS$DIV	Return `f1` / `f2`
FS$DIV_R	Return `f2` / `f1`
FS$INC	Return `f1` + 1.0
FS$INCR	Return `f1` + 1.0 (*Note:* `f1` is not popped.)
FS$DEC	Return `f1` - 1.0
FS$DECR	Return `f1` - 1.0 (*Note:* `f1` is not popped.)
FS$GE	Return `f1` $\geq$ `f2` (Return 1 or 0 in A8 and set status.)
FS$LE	Return `f1` $\leq$ `f2` (Return 1 or 0 in A8 and set status.)
FS$GT	Return `f1` > `f2` (Return 1 or 0 in A8 and set status.)
FS$LT	Return `f1` < `f2` (Return 1 or 0 in A8 and set status.)
FS$EQ	Return `f1` == `f2` (Return 1 or 0 in A8 and set status.)
FS$NE	Return `f1` != `f2` (Return 1 or 0 in A8 and set status.)
FS$ZERO	Returns 1 if `f1` = 0

5.7.4 Conversion Functions

Assume that:

- `f` and `d` are single-precision or double-precision floating-point values on the stack,
- `i` is an integer that is passed in A8, **and**
- `u` is an unsigned integer that is passed in A8.

Function	Action
FD$DTOF	Convert `d` to single precision and return on stack.
FD$DTOI	Convert `d` to a signed integer and return in A8.
FD$DTOU	Convert `d` to an unsigned integer and return in A8.
FD$FTOD	Convert `f` to double precision and return on stack.
FD$FTOI	Convert `f` to a signed integer and return in A8.
FD$FTOU	Convert `f` to an unsigned integer and return in A8.
FD$UTOD	Convert `u` to double precision and return on stack.
FD$ITOD	Convert `i` to double precision and return on stack.
FD$UTOF	Convert `u` to single precision and return on the stack.
FD$ITOF	Convert `i` to single precision and return on the stack.

5.7.5 Floating-Point Errors

You can customize this function in any way you wish; write your own function and use the archiver to include the function into the floating-point library.

Function	Action
—fp—error	Called whenever a floating-point exception occurs.

5.8 System Initialization

Before you can run a C program, the C runtime environment must be created. This task is performed by the C boot routine, which is a function called c_int00. The `boot.obj` module, which is a member of the `rts.lib` library, contains the c_int00 routine. (The source for the boot module is `boot.c` in the `rts.src` library.)

The c_int00 routine can be called by reset hardware to begin running the system. The `boot.obj` module must be combined with the C object modules at link time; this occurs by default when you use the -c or -cr linker options and include `rts.lib` as one of the linker input files. When C programs are linked, the linker sets the entry point value in the executable output module to the symbol `_c_int00`; this symbol points to the beginning of the c_int00 routine.

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

1) Reserves space in the .bss section for the program stack and the system stack and sets up the initial stack and frame pointers.

2) Autoinitializes global variables by copying the data from the initialization tables in .cinit to the storage allocated for the variables in .bss. (Note that in the RAM initialization model, a loader performs this step before the program runs – it is not performed by the boot routine.)

3) Calls the function `main` to begin running the C program.

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

5.8.1 Initializing the Stack

C code uses two stacks to manage the runtime environment:

- The **system stack** is used to save the status of a calling function or of an interrupted function. The system stack starts at the highest address in the stack space and grows toward lower addresses.

  The SP register is a dedicated register that points to the system stack. The TMS34010 instruction set supports several instructions for manipulating the system stack, including:

  - MMTM (save registers)
  - MMFM (restore registers)
  - CALLA or CALL (call a function)
  - RETS or RETI (return from a function or interrupt)

- The **program stack** is used for local frame generation during a function call; it is used for passing arguments to functions and for allocating local (temporary) variables for a called function. The program stack is controlled entirely through software. The TMS34010 does not dedicate a register to point to the program stack; however, the C compiler uses register A14 (STK) as the program-stack pointer.

The boot routine allocates memory for both stacks in the .bss section; this memory is allocated as a single, static array named SYS_STACK. The boot routine defines a constant named STACK_SIZE that determines the size of the SYS_STACK array; the default stack size is 2000 bytes. (For information about changing the stack size, see Section 5.1.2 on page 5-3.)

The two stacks share the space by growing towards each other from opposite sides of the array. A stack overflow occurs if the stacks collide; although there is no way to recover from a stack overflow, you can check for overflow conditions by invoking the code generator with the -x option.

5.8.2 Autoinitialization of Variables and Constants

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

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

> **Note:**
>
> In standard C, global and static variables which are not explicitly initialized are set to 0 before program execution. The TMS34010 C compiler does not adhere to this convention. Any variable which must have an initial value of 0 must be explicitly initialized.

There are two methods for copying the autoinitialization data into memory; these methods are called the ROM and RAM models of autoinitialization. The remainder of this section contains specific information about autoinitialization; Section 5.8.2.1 describes the format of the initialization tables, Section 5.8.2.2 describes the ROM model, and Section 5.8.2.3 describes the RAM model.

5.8.2.1 Initialization Tables

The tables in the .cinit section consist of initialization records of varying sizes. Figure 5-5 shows the format of the .cinit section and of an initialization record.

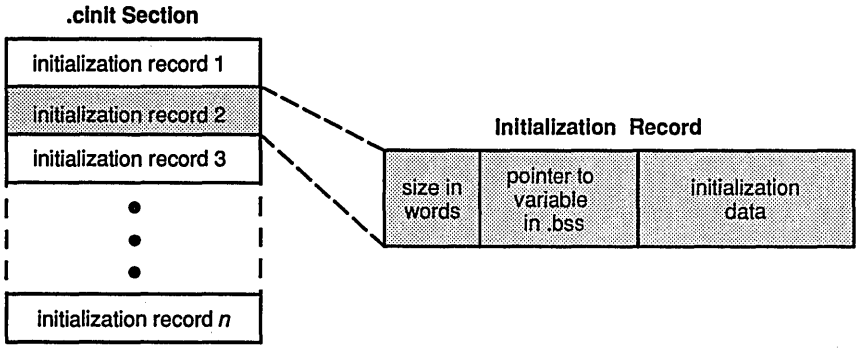

Figure 5-5. Format of Initialization Records in the .cinit Section

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

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

For example, suppose that 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:

```
.word  2            ; size in words of i
.long  _i           ; address of i in .bss
.long  23           ; 2 words of data for
                    ; initializing i
.word  10           ; size in words of a
.long  _a           ; address of a in .bss
.long  1,2,3,4,5    ; 10 words of data for
                    ; initializing a
```

The .cinit section must contain **only** initialization tables in this format. If you interface assembly language modules to your C program, do not use the .cinit section for any other purpose.

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

5.8.2.2 ROM Autoinitialization Model

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

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

Figure 5-6 illustrates the ROM model of autoinitialization.

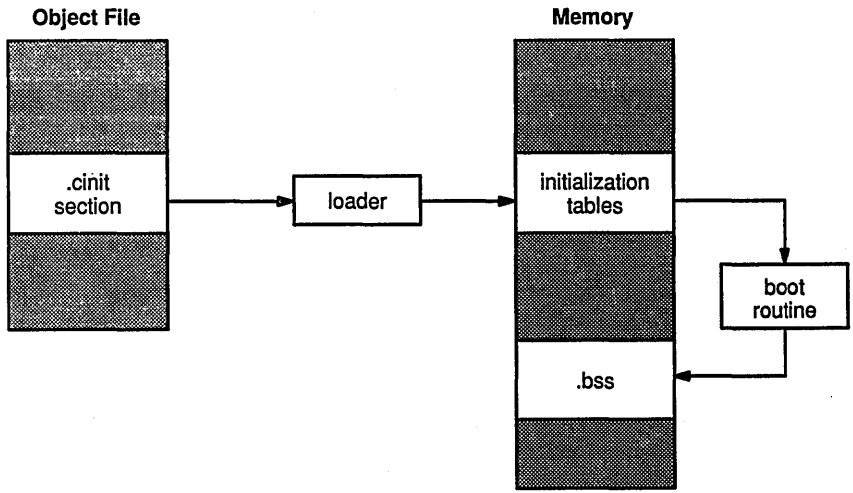

Figure 5-6. ROM Model of Autoinitialization

5.8.2.3 RAM Autoinitialization Model

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

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

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

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

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

Figure 5-7 illustrates the RAM model of autoinitialization.

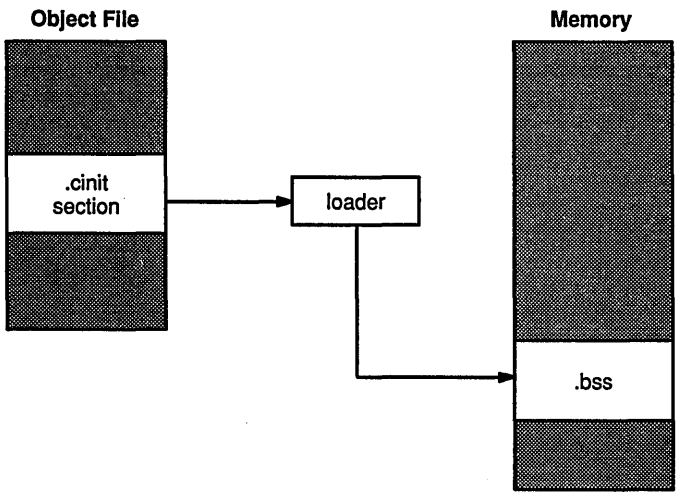

Figure 5-7. RAM Model of Autoinitialization

Section 6

Runtime-Support Functions

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

This section 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
6.1	Header Files	6-2
6.2	Summary of Runtime-Support Functions and Macros	6-9
6.3	Functions Reference	6-14

6.1 Header Files

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

- A set of related functions,
- 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       setjmp.h      string.h
float.h       stdarg.h      time.h
```

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

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

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

Section 6.1.1 through Section 6.1.10 describe the header files that are included with the TMS34010 C compiler. Section 6.2 (page 6-9) lists the functions that these headers declare.

6.1.1 Diagnostic Messages (assert.h)

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

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

The assert macro is defined as follows:

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

6.1.2 Character Typing and Conversion (ctype.h)

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

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

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

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

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

6.1.3 Limits (float.h and limits.h)

The `float.h` and `limits.h` headers define macros that expand to useful limits and parameters of the TMS34010's numeric representations. Table 6-1 and Table 6-2 list these macros and the limits they are associated with.

Table 6-1. Macros that Supply Integer Type Range Limits (limits.h)

Macro	Value	Description
CHAR_BIT	8	Number of bits in type char
SCHAR_MIN	-128	Minimum value for a signed char
SCHAR_MAX	127	Maximum value for a signed char
UCHAR_MAX	255	Maximum value for an unsigned char
CHAR_MIN	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	-2147483648	Minimum value for an int
INT_MAX	2147483647	Maximum value for an int
UINT_MAX	4294967295	Maximum value for an unsigned int
LONG_MIN	-2147483648	Minimum value for a long int
LONG_MAX	2147483647	Maximum value for a long int
ULONG_MAX	4294967295	Maximum value for an unsigned long int

Table 6-2. Macros that Supply Floating-Point Range Limits (float.h)

Macro	Value	Description
FLT_RADIX	2	Base or radix of exponent representation
FLT_ROUNDS	0	Rounding mode for floating-point addition (rounds to nearest integer)
FLT_DIG	6	Number of decimal digits of precision for a float
FLT_MANT_DIG	23	Number of base-FLT_RADIX digits in the mantissa of a float
FLT_MIN_EXP	-126	Smallest exponent (base 2) for a float
FLT_MAX_EXP	127	Largest exponent (base 2) for a float
FLT_EPSILON	1.1920929E-07F	Minimum positive float *x* such that 1.0 + *x* ≠ .1.0
FLT_MIN	5.8774720E-39F	Minimum positive float
FLT_MAX	1.7014116E+38F	Maximum positive float
FLT_MIN_10_EXP	-39	Minimum negative integer such that 10 raised to that power is in the range of normalized floats
FLT_MAX_10_EXP	38	Maximum positive integer such that 10 raised to that power is in the range of normalized floats
DBL_DIG LDBL_DIG	15	Number of decimal digits of precision for a double or a long double
DBL_MANT_DIG LDBL_MANT_DIG	52	Number of base-FLT_RADIX digits in the mantissa of a double or a long double
DBL_MIN_EXP LDBL_MIN_EXP	-1022	Smallest exponent (base 2) for a double or long double
DBL_MAX_EXP LDBL_MAX_EXP	1023	Largest exponent (base 2) for a double or long double
DBL_EPSILON LDBL_EPSILON	2.2204460492503131E-16	Minimum positive double or long double *x* such that 1.0 + *x* ≠ .1.0
DBL_MIN LDBL_MIN	1.1125369292536922E-308	Minimum positive double or long double
DBL_MAX LDBL_MAX	8.9884656743116201E+307	Maximum positive double or long double
DBL_MIN_10_EXP LDBL_MIN_10_EXP	-308	Minimum negative integer such that 10 raised to that power is in the range of normalized doubles or long doubles
DBL_MAX_10_EXP LDBL_MAX_10_EXP	307	Maximum positive integer such that 10 raised to that power is in the range of normalized doubles or long doubles

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

6.1.4 Floating-Point Math (math.h, errno.h)

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

The `math.h` header also defines three macros that can be used with the math functions for reporting errors:

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

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

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

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

When a function produces a floating-point return value that is too large to be represented, it returns HUGE_VAL instead.

6.1.5 Nonlocal Jumps (setjmp.h)

The `setjmp.h` header declares a function, a macro, and a type that are used for bypassing the normal function call and return conventions.

- The type that is declared is *jmp_buf*, which is an array type suitable for holding the information needed to restore a calling environment.

- The *setjmp* macro saves its calling environment in its `jmp_buf` argument, for later use by the *longjmp* function. The next invocation of longjmp, even in a different function, causes a jump back to the point at which setjmp was called.

6.1.6 Variable Arguments (stdarg.h)

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

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

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

6.1.7 Standard Definitions (stddef.h)

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

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

The macros include:

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

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

6.1.8 General Utilities (stdlib.h)

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

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

In TMS34010 C, ints and longs are the same size, so div_t and ldiv_t share a common definition:

```
typedef struct { int quot, rem; } div_t, ldiv_t;
```

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

- Memory management functions that allow you to allocate and deallocate packets of memory. The amount of memory that these functions can use is defined by the constant MEMORY_SIZE in the runtime-support module memory.c. (This module is archived in the file rts.src.) By default, the amount of memory available for allocation is 1000 bytes. You can change this amount by modifying MEMORY_SIZE 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.

- Program-exit functions that allow your program to terminate normally or abnormally.

- Integer-arithmetic that is not provided as a standard part of the C language.

6.1.9 String Functions (string.h)

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

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

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

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

6.1.10 Time Functions (time.h)

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

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

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

The header declares three types:

- *clock_t*, an arithmetic type that represents time,
- *time_t*, an arithmetic type that represents time, **and**
- *struct tm* is a structure that holds the components of calendar time, called *broken-down time*. The structure has the following members:

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

`tm_isdst` can have one of three values:

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

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

6.2 Summary of Runtime-Support Functions and Macros

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

Floating-Point Math Functions (Continued)	
Macro and Syntax	**Description**
`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 `x`
`double cosh(x)` `  double x;`	Returns the hyperbolic cosine of a floating-point value `x`
`double exp(x)` `  double x;`	Returns the exponential function of a real number `x`
`double fabs(x)` `  double x;`	Returns the absolute value of a floating-point value `x`
`double floor(x)` `  double x;`	Returns the largest integer less than or equal to `x`
`double fmod(x, y)` `  double x, y;`	Returns the floating-point remainder of `x/y`

`double frexp(value, exp)`
`  double value;`
`  int *exp;`
	Breaks 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 `x`
`double log10(x)` `  double x;`	Returns the base-10 logarithm of a real number `x`

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

`double pow(x, y)` `  double x, y;`	Returns `x` raised to the power `y`
`double sin(x)` `  double x;`	Returns the sine of a floating-point value `x`
`double sinh(x)` `  double x;`	Returns the hyperbolic sine of a floating-point value `x`
`double sqrt(x)` `  double x;`	Returns the nonnegative square root of a real number `x`
`double tan(x)` `  double x;`	Returns the tangent of a floating-point value `x`
`double tanh(x)` `  double x;`	Returns the hyperbolic tangent of a floating-point value `x`
Variable Argument Functions and Macros **Header File:** `stdarg.h`	
Function/Macro and Syntax	**Description**
type `va_arg(ap,` *type*`)` `  va_list ap;`	Accesses the next argument of type *type* in a variable-argument list
`void va_end(ap)` `  va_list ap;`	Resets the calling mechanism after using va_arg
`void va_start(ap,` *parmN*`)` `  va_list ap;`	Initializes `ap` to point to the first operand in the variable-argument list

General Utilities **Header File:** stdlib.h	
Function and Syntax	**Description**
int **abs**(j) int j;	Returns the absolute value of j
void **abort**()	Terminates a program abnormally
void **atexit**(fun) void (*fun)();	Registers the function point to by fun, to be called without arguments at normal program termination
double **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, each of size bytes
div_t **div**(numer, denom) int numer, denom	Divides numer by denom
void **exit**(status) int status;	Terminates a program normally
void **free**(ptr) void *ptr;	Deallocates memory space allocated by malloc, calloc, or realloc
long int **labs**(j) long int j;	Returns the absolute value of j
ldiv_t **ldiv**(numer, denom) long int numer, denom	Divides numer by denom
int **ltoa**(n, buffer) long n; char *buffer;	Converts n to the equivalent ASCII string
void ***malloc**(size) size_t size;	Allocates memory for an object of size bytes
void **minit**()	Resets all the memory previously allocated by malloc, calloc, or realloc
char ***movmem**(src,dest,count) char *src, *dest; int count;	Moves count bytes from 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
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;	Uses seed to reset the random number generator 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**(nptr, endptr, base) char *nptr, **endptr; int base;	Converts an ASCII string to an unsigned long integer

String Functions **Header File:** `string.h`	
Function and Syntax	**Description**

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

`int memcmp(s1, s2, n)`
`void *s1, *s2;`
`size_t n;`
	Compares the first `n` characters of `s1` to object 2

`void *memcpy(s1, s2, n)`
`void *s1, *s2;`
`size_t n;`
	Copies `n` characters from `s1` to object 2

`void *memmove(s1, s2, n)`
`void *s1, *s2;`
`size_t n;`
	Moves `n` characters from `s1` to object 2

`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 `s1` to the end of `s2`

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

`int strcmp(s1, s2)`
`char *s1, *s2;`
is greater than `s2`
	Compares strings and returns one of the following values:
	<0 if `s1` is less than `s2` =0 if `s1` is equal to `s2` >0 if `s1`

`int *strcoll(s1, s2)`
`char *s1, *s2;`
is greater than `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`

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

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

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

`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 `s1` of *any* character from `s2`

String Functions (continued) **Header File:** `string.h`	
Function and Syntax	**Description**

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

`size_t strspn(s1, s2)` `  char *s1, *s2;`	Returns the length of the initial segment of `s1`, which is entirely made up of characters from `s2`
`char *strstr(s1, s2)` `  char *s1, *s2;`	Finds the first occurrence of a 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
Time Functions **Header File:** `time.h`	
Function and Syntax	**Description**
`char *asctime(timeptr)` `  struct tm *timeptr;`	Converts a time to a string
`clock_t clock()`	Determines the processor time used
`char *ctime(timeptr)` `  struct tm *timeptr;`	Converts calendar time to local time
`double difftime(time1,time0)` `  time_t time1, tim0;`	Returns the difference between two calendar times
`struct tm *gmtime(timer)` `  time_t *timer;`	Converts calendar time Greenwich Mean Time
`struct tm *localtime(timer)` `  time_t *timer;`	Converts calendar time to local time
`time_t mktime(timeptr)` `  struct tm *timeptr;`	Converts local time to calendar time

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

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

6.3 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	6-16
abs	6-17
acos	6-18
asctime	6-19
asin	6-20
assert	6-21
atan	6-22
atan2	6-23
atexit	6-24
atof	6-25
atoi	6-25
atol	6-25
bsearch	6-26
calloc	6-27
ceil	6-28
clock	6-29
cos	6-30
cosh	6-31
ctime	6-32
difftime	6-33
div	6-34
exit	6-35
exp	6-36
fabs	6-37
floor	6-38
fmod	6-39
free	6-40
frexp	6-41
gmtime	6-42
isalnum	6-43
isalpha	6-43
isascii	6-43
iscntrl	6-43
isdigit	6-43
isgraph	6-43
islower	6-43
isprint	6-43
ispunct	6-43
isspace	6-43
isupper	6-43
isxdigit	6-43
labs	6-17
ldexp	6-44
ldiv	6-34
localtime	6-45

log 6-46
log10 6-47
ltoa 6-48
malloc 6-49
memchr 6-50
memcmp 6-51
memcpy 6-52
memmove 6-53
memset 6-54
minit 6-55
mktime 6-56
modf 6-57
movmem 6-58
pow 6-59
qsort 6-60
rand 6-61
realloc 6-62
sin 6-63
sinh 6-64
sqrt 6-65
srand 6-61
strcat 6-66
strchr 6-67
strcmp 6-68
strcoll 6-69
strcpy 6-69
strcspn 6-70
strerror 6-71
strftime 6-72
strlen 6-73
strncat 6-74
strncmp 6-75
strncpy 6-76
strpbrk 6-77
strrchr 6-78
strspn 6-79
strstr 6-80
strtod 6-81
strtok 6-82
strtol 6-81
strtoul 6-81
tan 6-83
tanh 6-84
time 6-85
toascii 6-86
tolower 6-87
toupper 6-87
va_arg 6-88
va_end 6-88
va_start 6-88

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 TMS34010 implementation of the abort function calls the exit function with a value of 0, and is defined as follows:

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

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

Syntax

```
#include  <stdlib.h>

int  abs(j)
     int  j;

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

Defined in

`abs.c` in `rts.src`

Description

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

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

Since *int* and *long int* are functionally equivalent types in TMS34010 C, the abs and labs functions are also functionally equivalent.

Syntax

```
#include  <math.h>

double acos(x)
   double x;
```

Defined in

asin.obj in rts.lib

Description

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

Example

```
double realval, radians;

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

Syntax

```
#include  <time.h>

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

Defined in

`actime.c` in `rts.src`

Description

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

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

The function returns a pointer to the converted string.

For more information about the functions and types that the `time.h` header declares, see Section 6.1.10 on page 6-7.

Syntax

```
#include <math.h>

double asin(x)
   double x;
```

Defined in

`asin.obj` in `rts.lib`

Description

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

Example

```
double realval, radians;

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

Syntax

```
#include  <assert.h>

void  assert(expression)
    int   expression;
```

Defined in `assert.h` as macros

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

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

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

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

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

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

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

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

Syntax

```
#include  <math.h>

double atan(x)
   double x;
```

Defined in

atan.obj in rts.lib

Description

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

Example

```
double realval, radians;

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

Syntax

```
#include  <math.h>

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

Defined in

atan.obj in rts.lib

Description

The atan2 function returns the inverse tangent of y/x. The function uses the signs of the arguments to determine the quadrant of the return value. Both arguments cannot be 0. The return value is an angle in the range [-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;
```

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

Description Three functions convert ASCII strings to numeric representations:

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

  [space] [sign] digits [.digits] [e|E [sign] integer]

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

  [space] [sign] digits

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

  [space] [sign] digits

The *space* is indicated by a space (from the space bar), a horizontal or vertical tab, a carriage return, a form feed, or a newline. Following the space is an optional *sign*, and then *digits* that represent the integer portion of the number. The fractional part of the number follows, then the exponent, including an optional *sign*.

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

Since *int* and *long* are functionally equivalent in TMS34010 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 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>

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 change the value of MEMORY_SIZE and reassembling memory.c. For more information, see Section 5.1.3, Dynamic Memory Allocation, on page 5-4.

Example

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

```
ptr = calloc(10,2) ;  /* Allocate and clear 10 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 <time.h>
clock_t clock()
```

Defined in

`clock.c` in `rts.src`

Description

The clock function determines the amount of processor time used. It returns an approximation of the processor time used by a program since the program began running. The time in seconds is the return value divided by the value of the macro CLK_TCK.

If the processor time is not available or cannot be represented, the clock function returns the value of (clock_t)-1.

> **Note:**
>
> The clock function is target-system specific, so you must write your own clock function. You must also define the CLK_TCK macro according to the granularity of your clock, so that the value returned by `clock()` (number of clock ticks) can be divided by CLK_TCK to produce a value in seconds.

For more information about the functions and types that the `time.h` header declares, see Section 6.1.10 on page 6-7.

Syntax

```
#include  <math.h>

double cos(x)
   double x;
```

Defined in

sin.obj in rts.lib

Description

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

Example

```
double radians, cval;    /* cos returns cval */

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

Syntax

```
#include  <math.h>

double cosh(x)
   double x;
```

Defined in

sinh.obj in rts.lib

Description

The cosh function returns the hyperbolic cosine of a floating-point number, x. A range error occurs if the magnitude of the argument is too large.

Example

```
double x, y;

x = 0.0;
y = cosh(x);        /* return value = 1.0 */
```

Syntax

```
#include  <time.h>

char  *ctime(timer)
   time_t  *timer;
```

Defined in

`ctime.c` in `rts.src`

Description

The ctime function converts a calendar time (pointed to by `timer`) to local time, in the form of a string. This is equivalent to:

```
asctime(localtime(timer))
```

The function returns the pointer returned by the asctime function with that broken-down time as an argument.

For more information about the functions and types that the `time.h` header declares, see Section 6.1.10 on page 6-7.

Syntax

```
#include <time.h>

double difftime(time1, time0)
   time_t time1, time0;
```

Defined in `difftime.c` in `rts.src`

Description The difftime function calculates the difference between two calendar times, `time1` minus `time0`. The return value expresses seconds.

For more information about the functions and types that the `time.h` header declares, see Section 6.1.10 on page 6-7.

Syntax

```
#include  <stdlib.h>

div_t  div(numer, denom)
   int  numer, denom;

ldiv_t ldiv(numer, denom)
   long int  numer, denom;
```

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

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

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

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

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

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

If the division produces a remainder, then the sign of the quotient is the same as the algebraic quotient, and the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient.

Since ints and longs are equivalent types in TMS34010 C, these functions are also equivalent.

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 functions (without arguments) that were registered in reverse order of their registration.

The exit function does 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, see Section 5.1.3, Dynamic Memory Allocation, on page 5-4.

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 [½,1) or 0, so that `value` = x × 2(**`exp`). The frexp function stores the power in the int pointed to by `exp`. If `value` is 0, both parts of the result are 0.

Example

```
double fraction;
int exp;

fraction = frexp(3.0, &exp);

/* after execution, fraction is .75 and exp is 2 */
```

Syntax

```
#include  <time.h>

struct tm  *gmtime(timer)
   time_t  *timer;
```

Defined in

`gmtime.c` in `rts.src`

Description

The gmtime function converts a calendar time (pointed to by `timer`) into a broken-down time which is expressed as Greenwich Mean Time.

For more information about the functions and types that the `time.h` header declares, see Section 6.1.10 on page 6-7.

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 non-space character.
islower identifies lowercase alphabetic ASCII characters.
isprint identifies printable ASCII characters, including spaces (ASCII characters 32–126).
ispunct identifies ASCII punctuation characters.
isspace identifies ASCII spacebar, tab (horizontal or vertical), carriage return, formfeed, and newline characters.
isupper identifies uppercase ASCII alphabetic characters.
isxdigit identifies hexadecimal digits (0–9, a–f, A–F).

The C compiler also supports a set of macros that perform these same functions. The macros have the same names as the functions, but are prefixed with an underscore; for example, *_isascii* is the macro equivalent of the isascii function. In general, the macros execute more efficiently than the functions.

Syntax

```
#include  <math.h>

double ldexp(x, exp)
    double x;
    int    exp;
```

Defined in

ldexp.obj in rts.lib

Description

The ldexp function multiplies a floating-point number by a power of 2 and returns $x \times 2^{exp}$. exp can be a negative or a positive value. A range error may occur if the result is too large.

Example

```
double result;

result = ldexp(1.5, 5);   /* result is 48.0 */

result = ldexp(6.0, -3);  /* result is 0.75 */
```

Syntax

```
#include  <time.h>

struct tm  *localtime(timer)
   time_t  *timer;
```

Defined in

`localtime.c` in `rts.src`

Description

The local time function converts a calendar time (pointed to by `timer`) into a broken-down time which is expressed as local time. The function returns a pointer to the converted time.

For more information about the functions and types that the `time.h` header declares, see Section 6.1.10 on page 6-7.

Syntax

```
#include  <math.h>

double log(x)
   double x;
```

Defined in

log.obj.c 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.c 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 change the value of MEMORY_SIZE and reassembling memory.c. For more information, see Section 5.1.3, Dynamic Memory Allocation, on page 5-4.

Syntax

```
#include  <string.h>

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

Defined in

memchr.c in rts.src

Description

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

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

Syntax

```
#include <string.h>

int  memcmp(s1, s2, n)
    void   *s1, *s2;
    size_t n;
```

Defined in

memcmp.c in rts.src

Description

The memcmp function compares the first n characters of the object that s2 points to with the object that s1 points to. The function returns one of the following values:

< 0	if *s1 is less than *s2.
0	if *s1 is equal to *s2.
> 0	if *s1 is greater than *s2.

The memcmp function is similar to strncmp, except the objects that memcmp compares can contain values of 0.

Syntax

```
#include  <string.h>

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

Defined in

memmov.c in rts.src

Description

The memcpy function copies n characters from the object that s2 points to into the object that s1 points to. *If you attempt to copy characters of overlapping objects, the function's behavior is undefined.* The function returns the value of s1.

The memcpy function is similar to strncpy, except the objects that memcpy copies can contain values of 0.

Syntax

```
#include <string.h>

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

Defined in

memmov.c in rts.src

Description

The memmove function moves n characters from the object that s2 points to into the object that s1 points to; the function returns the value of s1. *The memmove function correctly copies characters between overlapping objects.*

Syntax

```
#include  <string.h>

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

Defined in

`memset.c` in `rts.src`

Description

The memset function copies the value of `c` into the first `n` characters of the object that `s` points to. The function returns the value of `s`.

Syntax

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

Defined in

`memory.c` in `rts.src`

Description

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

> **Note:**
>
> Calling the minig 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 change the value of `MEMORY_SIZE` and reassembling `memory.c`. For more information, see Section 5.1.3, Dynamic Memory Allocation, on page 5-4.

Syntax

```
#include  <time.h>

time_t  *mktime(timeptr)
   struct tm  *timeptr;
```

Defined in

mktime.c in rts.src

Description

The mktime function converts a broken-down time, expressed as local time, into proper calendar time. The timeptr argument points to a structure that holds the broken-down time.

The function ignores the original values of tm_wday and tm_yday, and does not restrict the other values in the structure. After successful completion, tm_wday and tm_yday are set appropriately, and the other components in the structure have values within the restricted ranges. The final value of tm_mday is not sent until tm_mon and tm_year are determined.

The return value is encoded as a value of type time_t. If the calendar time cannot be represented, the function returns the value -1.

Example

This example determines the day of the week that July 4, 2001, falls on.

```
#include  <time.h>
static const char *const wday[] = {
             "Sunday", "Monday", "Tuesday", "Wednesday",
             "Thursday", "Friday", "Saturday" };

struct tm time_str;

time_str.tm_year  = 2001 - 1900;
time_str.tm_mon   = 7;
time_str.tm_mday  = 4;
time_str.tm_hour  = 0;
time_str.tm_min   = 0;
time_str.tm_sec   = 1;
time_str.tm_isdst = 1;

mktime(&time_str);

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

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

For more information about the functions and types that the time.h header declares, see Section 6.1.10 on page 6-7.

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

Syntax

```
#include  <stdlib.h>

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

Defined in

qsort.c in rts.src

Description

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

This function sorts the array in ascending order.

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

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

The cmp function compares the objects that ptr1 and ptr2 point to, and returns one of the following values:

< 0	if *ptr1 is less than *ptr2.
0	if *ptr1 is equal to *ptr2.
> 0	if *ptr1 is greater than *ptr2.

Syntax

```
#include <stdlib.h>
int rand( )
void srand(seed)
    unsigned int seed;
```

Defined in

rand.c in rts.src

Description

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

- The **rand** function returns pseudo-random integers in the range 0–RAND_MAX.
- The **srand** function sets the value of seed so that a subsequent call to the rand function produces a new sequence of pseudo-random numbers. The srand function does not return a value.

If you call rand before calling srand, rand generates the same sequence it would produce if you first called srand with a seed value of 1. If you call srand with the same seed value, rand generates the same sequence of numbers.

Syntax

```
#include <stdlib.h>

void *realloc(ptr, size)
    void   *ptr;   /* pointer to object to change   */
    size_t size;   /* new size (in bytes) of packet */
```

Defined in

memory.c in rts.src

Description

The realloc function changes the size of the allocated memory pointed to by ptr, to the size specified in bytes by size. The contents of the memory space (up to the lesser of the old and new sizes) is not changed.

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

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

The memory that realloc uses is in a special memory pool or heap. 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 change the value of MEMORY_SIZE and reassembling memory.c. For more information, see Section 5.1.3, Dynamic Memory Allocation, on page 5-4.

Syntax

```
#include  <math.h>

double sin(x)
   double x;
```

Defined in

sin.obj in rts.lib

Description

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

Example

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

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

Syntax

```
#include  <math.h>

double sinh(x)
   double x;
```

Defined in

sinh.obj in rts.lib

Description

The sinh function returns the hyperbolic sine of a floating-point number, x. A range error occurs if the magnitude of the argument is too large.

Example

```
double x, y;

x = 0.0;
y = sinh(x);        /* return value = 0.0 */
```

Syntax

```
#include  <math.h>

double sqrt(x)
   double x;
```

Defined in

sqrt.obj in rts.lib

Description

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

Example

```
double x, y;

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

Syntax

```
#include <string.h>

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

Defined in

`strcat.c` in `rts.src`

Description

The strcat function appends a copy of `s2` to (including a terminating null character) to the end of `s1`. The initial character of `s2` overwrites the null character that originally terminated `s1`. The function returns the value of `s1`.

Syntax

```
#include  <string.h>

char *strchr(s, c)
    char *s;
    char c;
```

Defined in `strchr.c` in `rts.src`

Description The strchr function finds the first occurrence of `c` (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;
```

Defined in

`strcmp.c` in `rts.src`

Description

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

The functions return one of the following values:

< 0	if `*s1` is less than `*s2`.
0	if `*s1` is equal to `*s2`.
> 0	if `*s1` is greater than `*s2`.

Syntax

```
#include  <string.h>

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

Defined in

`strcpy.c` in `rts.src`

Description

The strcpy function copies `s2` (including a terminating null character) into `s1`. If you attempt to copy strings that overlap, the function's behavior is undefined. The function returns a pointer to the destination string.

Syntax

```
#include  <string.h>

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

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

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

Syntax

```
#include  <string.h>

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

Defined in

`strerror.c` in `rts.src`

Description

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

Syntax

```
#include  <time.h>

size_t  *strftime(s, maxsize, format, timeptr)
   char    *s, *format;
   size_t  maxsize;
   struct tm  *timeptr;
```

Defined in `strftime.c` in `rts.src`

Description The strftime function formats a time (pointed to by `timeptr`) according to a `format` string, and returns the formatted time in the string `s`. Up to `maxsize` characters can be written to `s`. The `format` parameter is a string of characters that tells the strftime function how to format the time; the following list shows the valid characters and describes what each character expands to.

Character is replaced by ...

%a	the locale's abbreviated weekday name
%A	the locale's full weekday name
%b	the locale's abbreviated month name
%B	the locale's full month name
%c	the locale's appropriate date and time representation
%d	the day of the month as a decimal number (0–31)
%H	the hour (24-hour clock) as a decimal number (00–23)
%I	the hour (12-hour clock) as a decimal number (01–12)
%j	the day of the year as a decimal number (001–366)
%m	the month as a decimal number (01–12)
%M	the minute as a decimal number (00–59)
%p	the locale's equivalent of either AM or PM
%S	the second as a decimal number (00–50)
%U	the week number of the year (Sunday is the first day of the week) as a decimal number (00–52)
%x	the locale's appropriate date representation
%X	the locale's appropriate time representation
%y	the year without century as a decimal number (00–99)
%Y	the year with century as a decimal number
%Z	the time zone name, or by no characters if no time zone exists
%%	%

For more information about the functions and types that the `time.h` header declares, see Section 6.1.10 on page 6-7.

Syntax

```
#include  <string.h>

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

Defined in

`strlen.c` in `rts.src`

Description

The strlen function returns the length of `s`. In C, a character string is terminated by the first byte with a value of 0 (a null character). The returned result does not include the terminating null character.

Syntax

```
#include <string.h>

char *strncat(s1, s2, n)
    char   *s1, *s2;
    size_t n;
```

Defined in

`strncat.c` in `rts.src`

Description

The strncat function appends up to `n` characters of `s2` (including a terminating null character) to the end of `s1`. The initial character of `s2` overwrites the null character that originally terminated `s1`; strncat appends a null character to result. The function returns the value of `s1`.

Syntax

```
#include  <string.h>

int  strncmp(s1, s2, n)
    char   *s1, *s2;
    size_t n;
```

Defined in strncmp.c in rts.src

Description The strncmp function compares up to n characters of s2 with s1. The function returns one of the following values:

< 0	if *s1 is less than *s2.
0	if *s1 is equal to *s2.
> 0	if *s1 is greater than *s2.

Syntax

```
#include <string.h>

char *strncpy(s1, s2, n)
    char   *s1, *s2;
    size_t n;
```

Defined in

strncpy.c in rts.src

Description

The strncpy function copies up to n characters from s2 into s1. If s2 is n characters long or longer, the null character that terminates s2 is not copied. If you attempt to copy characters from overlapping strings, the function's behavior is undefined. If s2 is shorter than n characters, strncpy appends null characters to s1 so that s1 contains n characters. The function returns the value of s1.

Syntax

```
#include  <string.h>

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

Defined in

`strpbrk.c` in `rts.src`

Description

The strpbrk function locates the first occurrence in `s1` of *any* character in `s2`. If strpbrk finds a matching character, it returns a pointer to that character; otherwise, it returns a null pointer (0).

Syntax

```
#include  <string.h>

char *strrchr(s ,c)
    char *s;
    int  c;
```

Defined in

`strrchr.c` in `rts.src`

Description

The strrchr function finds the last occurrence of `c` in `s`. If strrchr finds the character, it returns a pointer to the character; otherwise, it returns a null pointer (0).

Syntax

```
#include  <string.h>

size_t *strspn(s1, s2)
    int   *s1, *s2;
```

Defined in `strspn.c` in `rts.src`

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

Syntax

```
#include <string.h>

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

Defined in `strstr.c` in `rts.src`

Description The strstr function finds the first occurrence of `s2` in `s1` (excluding the terminating null character). If strstr finds the matching string, it returns a pointer to the located string; if it doesn't find the string, it returns a null pointer. If `s2` points to a string with length 0, then strstr returns `s1`.

Syntax

```
#include  <stdlib.h>

double strtod(nptr, endptr)
    char  *nptr;
    char  **endptr;

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

unsigned long int strtoul(nptr, endptr, base)
    char  *nptr;
    char  **endptr;
    int   base;
```

Defined in

`strtod.c` in `rts.src`
`strtol.c` in `rts.src`
`strtoul.c` in `rts.src`

Description

Three functions convert ASCII strings to numeric values. For each function, argument `nptr` points to the original string. Argument `endptr` points to a pointer; the functions set this pointer to point to the first character after the converted string. The functions that convert to integers also have a third argument, `base`.

- The **strtod** function converts a string to a floating-point value. The string must have the following format:

  [space] [sign] digits [.digits] [e|E [sign] integer]

  The function returns the converted string; if the original string is empty or does not have the correct format, the function returns a 0. If the converted string would cause an overflow, the function returns ±HUGE_VAL; if the converted string would cause an underflow, the function returns 0. If the converted string causes an overflow or an underflow, errno is set to the value of ERANGE.

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

  [space] [sign] digits [.digits] [e|E [sign] integer]

- The **strtoul** function converts a string to a long integer. The string must be specified in the following format:

  [space] [sign] digits [.digits] [e|E [sign] integer]

The *space* is indicated by a spacebar, horizontal or vertical tab, carriage return, form feed, or newline. Following the space is an optional *sign*, and then *digits* that represent the integer portion of the number. The fractional part of the number follows, then the exponent, including an optional *sign*.

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

Syntax

```
#include <string.h>

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

Defined in

`strtok.c` in `rts.src`

Description

Successive calls to the strtok function break `s1` into a series of tokens, each delimited by a character from `s2`. Each call returns a pointer to the next token.

Syntax

```
#include  <math.h>

double tan(x)
   double x;
```

Defined in

tan.obj in rts.lib

Description

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

Example

```
double x, y;

x = 3.1415927/4.0;
y = tan(x);           /* return value = 1.0 */
```

Syntax

```
#include  <math.h>

double tanh(x)
   double x;
```

Defined in

tanh.obj in rts.lib

Description

The tanh function returns the hyperbolic tangent of a floating-point number, x.

Example

```
double x, y;

x = 0.0;
y = tanh(x);        /* return value = 0.0 */
```

Syntax

```
#include  <time.h>

time_t  time(timer)
   time_t  *timer;
```

Defined in

`time.c` in `rts.src`

Description

The time function determines the current calendar time, represented in seconds. If the calendar time is not available, the function returns -1. If `timer` is not a null pointer, the function also assigns the return value to the object that `timer` points to.

For more information about the functions and types that the `time.h` header declares, see Section 6.1.10 on page 6-7.

> **Note:**
>
> The time function is target-system specific, so you must write your own time function.

Syntax

```
#include  <ctype.h>

int  toascii(c)
    char  c;
```

Defined in `toascii.c` in `rts.src`

Description The toascii function ensures that `c` is a valid ASCII character by masking the lower seven bits. There is also a toascii macro.

Syntax

```
#include <ctype.h>

int tolower(c)
    char c;

int toupper(c)
    char c;
```

Defined in

`tolower.c` in `rts.src`
`toupper.c` in `rts.src`

Description

Two functions convert the case of a single alphabetic character, `c`, to upper or lower case:

- The **tolower** function converts an uppercase argument to lowercase. If `c` is 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

Error Messages

Compiler error messages are displayed in the following format, which shows the line number in which the error occurs and the text of the message:

"name.c", line *n* : *error message*

These types of errors are not fatal.

The errors listed below cause the compiler to abort immediately.

- **>> cannot allocate sufficient memory**

  The compiler requires a minimum of 512K bytes of memory to run; this message indicates that this amount is not available. Supply more dynamic RAM.

- **>> can't open "filename" as source**

  The compiler cannot find the file name as entered. Check for spelling errors and check to see that the named file actually exists.

- **>> can't open "filename" as intermediate file**

  The compiler cannot create the output file. This is usually caused by either an error in the syntax of the filename or a full disk.

- **>> illegal extension "ext" on output file**

  The intermediate file cannot have a ".c" extension.

- **>> fatal errors found: no intermediate file produced**

  This message is printed after an unsuccessful compilation. Correct the errors (other messages will indicate particular errors) and try compilation again.

- **>> cannot recover from earlier errors: aborting**

  An error has occurred that prevents the compiler from continuing.

Appendix B

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's 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* 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 the utility of #ifdef and #ifndef in an #if directive. Only these operators, integer constants, and names which are known by the preprocessor should be used in *constant-expression*. In particular, the *sizeof* operator should not be used.
  - The **#ifdef** directive tests to see if *name* is a defined constant. The code following an #ifdef directive (up to an #else or an #endif) is compiled if *name* is defined (by the #define directive) and it has not been undefined by the #undef directive.
  - The **#ifndef** directive tests to see if *name* is *not* a defined constant. The code following an #ifndef directive (up to an #else or an #endif) is compiled if *name* is not defined (by the #define directive) or if it was undefined by the #undef directive.

- The **#else** directive begins an alternate block of code that is compiled if:

  - The condition tested by #if is false.
  - The name tested by #ifdef is not defined.
  - The name tested by #ifndef is defined.

  Note that the #else portion of a conditional block is *optional*; if the #if, #ifdef, or#ifndef test is not successful, then the preprocessor continues with the code following the #endif.

- The **#endif** directive ends a conditional block. Each #if, #ifdef, and #ifndef directive must have a matching #endif. Conditional compilation sequences can be nested.

Syntax

#include *"filename"*

or

#include *<filename>*

Description

The #include directive tells the preprocessor to read source statements from another file. The preprocessor includes (at the point in the code where #include is encountered) the contents of the *filename*, which are then processed. You can enclose the *filename* in double quotes or in angle brackets.

The *filename* can be a complete pathname or a filename with no path information.

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

> **Note:**
>
> If you enclose the *filename* in angle brackets, the preprocessor *does not* search for the file in the current direc'

For more information about the -i option and the environment variable, read Section 3.1.3 on page 3-4.

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

-a (code generator option) 3-9
abort 6-16
abort macro 6-16
abs function 6-17
acos function 6-18
alternate directories 3-4
archiver 1-3, 3-16
array alignment 5-5
ASCII conversion functions 6-25
asctime macro 6-19
asin macro 6-20
asm 4-2
assembler 1-3, 3-11, 3-12
assert macro 6-21, 6-2
assert.h header 6-2, 6-9
atan function 6-22
atan2 function 6-23
atexit function 6-24
atof 6-25
atof function 6-25
atoi 6-25
atoi function 6-25
atol 6-25
atol function 6-25
autoinitialization 3-13-3-16
  RAM model 3-13-3-16
  ROM model 3-13-3-16

B

batch files 3-11, 3-12
boot.obj 3-13, 3-15
broken-down time 6-8
bsearch function 6-26

C

-c (preprocessor option) 3-2
C compiler 1-3
-c option (linker) 3-13-3-16
calendar time 6-8
calloc function 6-27
C_DIR (environment variable) 3-4, 3-6
ceil function 6-28
character constants 4-2
character conversions 4-6
character typing/conversion
 functions 6-3, 6-9
  isalnum 6-43
  isalpha 6-43
  isascii 6-43
  iscntrl 6-43
  isdigit 6-43
  isgraph 6-43
  islower 6-43
  isprint 6-43
  ispunct 6-43
  isspace 6-43
  isupper 6-43
  isxdigit 6-43
.cinit section 3-15
_c_int00 3-13, 3-15
CLK_TCK macro 6-29, 6-8
clock function 6-29
clock_t type 6-8
code generator 3-8
  gspcg 3-8
  invocation 3-8
  options 3-9
    -a 3-9
    -o 3-9
    -q 3-9
    -r 3-9
    -s 3-9
    -v 3-9
    -x 3-9
    -z 3-9

compiler operation 3-1-3-16
constants
character 4-2
enumeration 4-2
floating-point 4-2
integer 4-2
cos function 6-30
cosh function 6-31
-cr option (linker) 3-13-3-16
ctime function 6-32
ctype.h header 6-3, 6-9

D

-d (preprocessor option) 3-2
daylight savings time 6-8
#define directive B-2
diagnostic messages 6-2
assert 6-21
NDEBUG 6-21
difftime function 6-33
div function 6-34
div_t type 6-6

E

EDOM macro 6-5
#else directive B-3
#endif directive B-3
entry points
_c_int00 3-13, 3-15
for C code 3-13, 3-15
reset vector 3-13
enum 4-2
enumeration constants 4-2
environment variable 3-4
EPROM programmers 1-3
ERANGE macro 6-5
error messages A-1
exit function 6-35
exp function 6-36
explicit pointer conversions 4-7

F

fabs function 6-37
fatal errors A-1
field manipulation 5-5
float.h header 6-3, 6-4
floating-point constants 4-2
floating-point conventions 5-17
floating-point conversions 4-6
floating-point math functions 6-5, 6-9
acos 6-18
asin 6-20
atan 6-22
atan2 6-23
ceil 6-28
cos 6-30
cosh 6-31
exp 6-36
fabs 6-37
floor 6-38
fmod 6-39
frexp 6-41
ldexp 6-44
log 6-46
log10 6-47
pow 6-59
sin 6-63
sinh 6-64
sqrt 6-65
tan 6-83
tanh 6-84
floor function 6-38
fmod function 6-39
font library 1-4
FP 5-4
frame pointer 5-4
free function 6-40
frexp function 6-41
function call conventions 5-8

G

general utility functions 6-6
abs 6-17
bsearch 6-26
div 6-34
labs 6-17
ldiv 6-34
qsort 6-60
rand 6-61
srand 6-61
global variables 4-10
gmtime function 6-42
gregorian time 6-8
gspcc 3-6
gspcg 3-8
gspcpp 3-2

H

hardware requirements (PC systems) 2-2
header files 6-2-6-8
HUGE_VAL macro 6-5

I

-i (preprocessor option) 3-3
-i option (preprocessor) 3-4
identifiers 4-2
#if directive B-3
#ifdef directive B-3
#ifndef directive B-3
#include 3-4
#include directive B-5
include files 3-4
inline assembly construct (asm) 4-11
installation 2-1
Macintosh/MPW 2-5
PCs 2-2
System V 2-4
Ultrix 2-4
VMS 2-3
instruction set 1-4
integer constants 4-2
integer conversions 4-6
integer expression analysis 5-17
integer return values 5-6
interfacing C with assembly language 5-12
assembly language modules 5-12
defining variables in assembly language 5-14
interrupt handling 5-16
invoking...
batch files 3-12
invoking the...
assembler 3-11
batch files 3-11
code generator 3-8
linker 3-13
parser 3-6
preprocessor 3-2
isalnum function 6-43
isalpha function 6-43
isascii function 6-43
iscntrl function 6-43
isdigit function 6-43
isgraph function 6-43
islower function 6-43
isprint function 6-43
ispunct function 6-43
isspace function 6-43
isupper function 6-43
isxdigit function 6-43

K

Kernighan and Ritchie
preprocessor 3-2
support tools 1-1
The C Programming Language 1-1, 1-4
keywords 4-2

L

labs function 6-17
ldexp function 6-44
ldiv function 6-34
ldiv_t type 6-6
limits
    floating-point types 6-3, 6-4
    integer types 6-3
limits.h header 6-3
#line directive B-6
linker 1-3, 3-13-3-16
linker command file 3-14
linking C code 3-13-3-16
listing files (assembler) 3-12
local time 6-8
localtime function 6-45
log function 6-46
log10 function 6-47
ltoa function 6-48

M

Macintosh/MPW 2-5
malloc function 6-49
math.h header 6-5, 6-9
memchr function 6-50
memcmp function 6-51
memcpy function 6-52
memmove function 6-53
memory management functions 6-27
    calloc 6-27
    free 6-40
    malloc 6-49
    minit 6-55
    movmem 6-58
    realloc 6-62
memory model 5-2
memset function 6-54
minit function 6-55
mktime function 6-56
modf 6-57
modf function 6-57
movmem function 6-58

N

NDEBUG macro 6-21, 6-2
nonlocal-jump functions 6-5
NULL macro 6-6

O

-o (code generator option) 3-9
object alignment 4-6
object format converter 1-3
object libraries 3-13
offsetof macro 6-6
operation of the compiler 3-1-3-16

P

-p (preprocessor option) 3-3
parser (gspcc) 3-6
    invocation 3-6
    options 3-7
        -q 3-7
        -z 3-7
PC installation 2-2
pow function 6-59
predefined names 3-3
preprocessor (gspcpp) 3-2
    invocation 3-2
    options 3-2
        -c 3-2
        -d 3-2
        -i 3-3
        -p 3-3
        -q 3-3
preprocessor directives B-1-B-6
    directives
primary expressions 4-7
program stack 5-3
program termination functions 6-16
    atexit 6-24
    exit 6-35
ptrdiff_t type 6-6

Q

-q (code generator option) 3-9
-q (parser option) 3-7
-q (preprocessor option) 3-3
qsort function 6-60

R

-r (code generator option) 3-9
RAM model of
  autoinitialization 3-13-3-16
rand function 6-61
realloc function 6-62
register conventions 5-6
register variables 4-8, 5-7
reserved registers 5-6
reset vector 3-13
ROM model of
  autoinitialization 3-13-3-16
rts.lib 3-13, 3-15, 6-1
rts.src 6-1
runtime environment 5-1-5-27
    interfacing C with assembly
      language 5-12
        assembly language
          modules 5-12
        defining variables in assembly lan-
          guage 5-14
runtime initialization 3-13
runtime support 3-13
runtime-support functions 6-1-6-88

S

-s (code generator option) 3-9
SDB 1-4
setjmp.h header 6-5
sin function 6-63
sinh function 6-64
size_t type 6-6
software development board 1-4
software installation 2-1
SP 5-3
sqrt function 6-65
srand function 6-61
stacks 5-3
static variables 4-10
stdarg.h header 6-6, 6-10
stddef.h header 6-6
stdlib.h header 6-6, 6-11
STK 5-3
strcat function 6-66
strchr function 6-67
strcmp function 6-68
strcoll function 6-68
strcpy function 6-69
strcspn function 6-70
strerror function 6-71
strftime function 6-72
string functions 6-7
    memchr 6-50
    memcmp 6-51
    memcpy 6-52
    memmove 6-53
    memset 6-54
    strcat 6-66
    strchr 6-67
    strcmp 6-68
    strcoll 6-68
    strcpy 6-69
    strcspn 6-70
    strerror 6-71
    strlen 6-73
    strncat 6-74
    strncmp 6-75
    strncpy 6-76
    strpbrk 6-77
    strrchr 6-78
    strspn 6-79
    strstr 6-80
    strtod 6-81
    strtok 6-82
    strtol 6-81
    strtoul 6-81
string.h header 6-7, 6-12, 6-13
strlen function 6-73
strncat function 6-74
strncmp function 6-75
strncpy function 6-76
strpbrk function 6-77
strrchr function 6-78
strspn function 6-79
strstr function 6-80
strtod function 6-81
strtok function 6-82

strtol function 6-81
strtoul function 6-81
structure packing 5-5
structures 4-7
style and symbol conventions 1-5
system initialization 5-22
system stack 5-3
System V installation 2-4

T

tan function 6-83
tanh function 6-84
time function 6-85
time functions 6-8
    asctime 6-19
    CLK_TCK 6-29
    clock 6-29
    ctime 6-32
    difftime 6-33
    gmtime 6-42
    localtime 6-45
    mktime 6-56
    strftime 6-72
    time 6-85
time.h header 6-8, 6-13
time_t type 6-8
tm structure 6-8
TMS34010 C language 4-1
toascii macro 6-86
    toascii 6-86
tolower function 6-87
toupper function 6-87
    tolower 6-87
    toupper 6-87

U

Ultrix installation 2-4
#undef directive B-2
unions 4-7

V

-v (code generator option) 3-9
va_arg macro 6-88
va_end function 6-88
va_list type 6-6
variable-argument functions 6-6, 6-88
    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
    va_arg 6-88
    va_end 6-88
    va_start 6-88
va_start macro 6-88
VAX installation 2-3
void 4-2

X

-x (code generator option) 3-9

Z

-z (code generator option) 3-9
-z (parser option) 3-7

TMS34010 C Compiler Reference Guide

We want to provide you with the best documentation possible – please help us by answering these questions and returning this card.

Is this manual adequate in helping you to use the TMS34010 C compiler for your application?

Were you familiar with the C programming language before you used the TMS34010 C compiler?

Are the references to other C-language documentation (i.e., Kernighan and Ritchie) adequate?

What kinds of examples would you like us to include in this manual?

How would you change this manual to make it more accurate or easier to use?

What information would you add to or delete from the Reference Card?

Additional comments:

Thank you for taking the time to fill out this card.

Your Name: _____

Company and Application: _____

Address: _____

_____

Would you like a reply? _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST-CLASS MAIL PERMIT NO. 6189 HOUSTON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

Product Documentation Services Manager
Texas Instruments Incorporated
P.O. Box 1443, M/S 640
Houston, Texas 77251–9879

TEXAS INSTRUMENTS

Printed in U.S.A., August 1988
1604903-9704

SPVU005A