TMS320C3x/C4x Code Generation Tools Getting Started Guide

Release 5.00

SPRU119B
March 1997

TEXAS INSTRUMENTS

Preface

Read This First

About This Manual

This manual tells you how to install release 5.00 of the TMS320C3x/C4x floating-point DSP code generation tools on your system. It also does the following:

- ❏ Tells you how to set environment variables for parameters that you use often
- ❏ Gets you started using the compiler, linker, and assembler
- ❏ Details the enhancements included and tells you where to find further information
- ❏ Describes how you can resolve problems that you may encounter on a PC™ running DOS (MS-DOS™ or PC-DOS™)

How to Use This Manual

The goal of this book is to get you started using the development tools specifically designed for the TMS320C3x/C4x floating-point DSPs. Following are the topics covered in this user's guide:

For information about …	**Refer to** …
Installing the code generation tools or setting environment variables on a PC running:	
MS–DOS, PC–DOS, or Windows 3.1	Chapter 1
Windows NT or Windows 95	Chapter 2
Installing the code generation tools or setting environment variables on a SPARCstation running SunOS	Chapter 3
Installing the code generation tools or setting environment variables on an HP 9000 Series 700 PA–RISC computer running HP–UX	Chapter 4
Getting started using the code generation tools	Chapter 5
Release notes	Chapter 6
Troubleshooting DOS systems	Appendix A
Accessing TMS320C3x peripherals through C pointers	Appendix B

Notational Conventions

This document uses the following conventions.

- ❑ Program listings, program examples, and interactive displays are shown in a `special typeface`. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold verson`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.). Here is an example of a command that you might enter:

  `mkdir` *`tool_dir`*

  In this example, you should type **`mkdir`** as shown and replace *`tool_dir`* with the name of your directory.

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

  .sfloat *value*

- ❑ Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you do not enter the brackets themselves. Here is an example of a command that has optional parameters:

  set DOS4GVM=[*option*[#*value*]] [*option*[#*value*]] ...

  This command allows you to specify one or more options and to indicate values with each option, if appropriate. In this example, you must include the # symbol if you enter a value.

Related Documentation From Texas Instruments

The following books describe the TMS320C3x/C4x floating-point DSPs and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C3x/C4x Assembly Language Tools User's Guide (literature number SPRU035) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the 'C3x and 'C4x generations of devices.

TMS320C3x/C4x Optimizing C Compiler User's Guide (literature number SPRU034) describes the TMS320 floating-point C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the 'C3x and 'C4x generations of devices.

TMS320C3x C Source Debugger User's Guide (literature number SPRU053) tells you how to invoke the 'C3x emulator, evaluation module, and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints. It also includes a tutorial that introduces basic debugger functionality.

TMS320C4x C Source Debugger User's Guide (literature number SPRU054) tells you how to invoke the 'C4x emulator and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints. It also includes a tutorial that introduces basic debugger functionality.

TMS320C3x User's Guide (literature number SPRU031) describes the 'C3x 32-bit floating-point microprocessor (developed for digital signal processing as well as general applications), its architecture, internal register structure, instruction set, pipeline, specifications, and DMA and serial port operation. Software and hardware applications are included.

TMS320C32 Addendum to the TMS320C3x User's Guide (literature number SPRU132) describes the TMS320C32 floating-point microprocessor (developed for digital signal processing as well as general applications). Discusses its architecture, internal register structure, specifications, and DMA and serial port operation. Hardware applications are also included.

TMS320C4x User's Guide (literature number SPRU063) describes the 'C4x 32-bit floating-point processor, developed for digital signal processing as well as parallel processing applications. Covered are its architecture, internal register structure, instruction set, pipeline, specifications, and operation of its six DMA channels and six communication ports.

Parallel Processing with the TMS320C4x (literature number SPRA031) describes parallel processing and how the 'C4x can be used in parallel processing. Also provides sample parallel processing applications.

TMS320C4x General-Purpose Applications User's Guide (literature number SPRU159) describes software and hardware applications for the 'C4x processor. Also includes development support information, parts lists, and XDS510 emulator design considerations.

TMS320C30 Evaluation Module Technical Reference (literature number SPRU069) describes board-level operation of the TMS320C30 EVM.

Digital Signal Processing Applications With the TMS320C30 Evaluation Module Selected Application Notes (literature number SPRA021) contains useful information for people who are preparing and debugging code. The book gives additional information about the TMS320C30 EVM, as well as C coding tips.

TMS320 Family Development Support Reference Guide (literature number SPRU011) describes the '320 family of digital signal processors and covers the various products that support this product line. This includes code-generation tools (compilers, assemblers, linkers, etc.) and system integration and debug tools (simulators, emulators, evaluation modules, etc.). Also covered are available documentation, seminars, the university program, and factory repair and exchange.

***Digital Signal Processing Applications with the TMS320 Family*, Volumes 1, 2, and 3** (literature numbers SPRA012, SPRA016, SPRA017) Volumes 1 and 2 cover applications using the 'C10 and 'C20 families of fixed-point processors. Volume 3 documents applications using both fixed-point processors as well as the 'C30 floating-point processor.

TMS320 DSP Designer's Notebook: Volume 1 (SPRT125). Presents solutions to common design problems using 'C2x, 'C3x, 'C4x, 'C5x, and other TI DSPs*.*

TMS320 Third-Party Support Reference Guide (literature number SPRU052) alphabetically lists over 100 third parties that provide various products that serve the family of '320 digital signal processors. A myriad of products and applications are offered—software and hardware development tools, speech recognition, image processing, noise cancellation, modems, etc.

Trademarks

If You Need Assistance . . .

❑ World-Wide Web Sites

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

❑ North America, South America, Central America

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

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

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

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

❑ Asia-Pacific

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

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

❑ Japan

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

❑ Documentation

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

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

Email: comments@books.sc.ti.com

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

Contents

Tables

Examples

Chapter 1

Setting Up the Tools on a PC Running DOS or Windows 3.1

This chapter helps you install release 5.00 of the TMS320C3x/C4x code generation tools and set up your code-development environment on a PC running MS-DOS, PC-DOS, or Windows™ 3.1. These tools include an optimizing C compiler and a full set of assembly language tools for developing and manipulating assembly language and object (executable) code.

The C compiler tools include:

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

The assembly language tools are composed of the following:

❑ Assembler
❑ Archiver
❑ Linker
❑ Absolute lister
❑ Cross-reference lister
❑ Hex conversion utility

Release 5.00 supports extended memory on DOS. Extended memory lets you compile or assemble large files that could not be built previously under DOS. Extended memory is enabled by the DOS/4GW™ memory extender from Tenberry Software, Inc. (formerly Rational Systems, Inc.), which is embedded in the TMS320C3x/C4x code generation tools package.

Topic **Page**

1.1 System Requirements

To install the TMS320C3x/C4x code generation tools on a PC, you need the following items:

- ❏ 80386, 80486, or Pentium-based PC running MS-DOS, PC-DOS, or Windows™ 3.1
- ❏ 4–16 Mbytes of free memory. 16 Mbytes is recommended to ensure optimum performance when compiling large C functions
- ❏ 10 Mbytes of disk space for the executable files and libraries
- ❏ CD-ROM drive

Note: Memory Needed

The code generation tools, when installed on a PC, require at least 4 Mbytes of memory, but you can expect some performance problems when using only 4 Mbytes. (16 Mbytes is recommended.) You may want to free as much memory as possible before installing the tools, especially if you have less than 16 Mbytes.

1.2 Installing the Tools

The code generation tools package is shipped on CD-ROM. The installation instructions vary according to your operating system.

Installing the tools on DOS systems

To install the tools on a DOS system, follow these steps:

1) Insert the TMS320C3x/C4x Software Toolkit CD-ROM into your CD-ROM drive.

2) Change to the CD-ROM drive (replace d with the name of your CD-ROM drive):

   `d:` ↵

3) Enter the following command:

   `install` ↵

4) Follow the on-screen instructions.

Installing the tools on Windows 3.1 systems

To install the tools on a Windows 3.1 system, perform the following steps:

1) Insert the TMS320C3x/C4x Software Toolkit CD-ROM into your CD-ROM drive.

2) Start Windows 3.1.

3) From the File menu, select Run.

4) In the dialog box, enter the following command (replace d with the name of your CD-ROM drive):

   `d:\setup.exe`

5) Click on OK.

6) Follow the on-screen instructions.

1.3 Setting Up the Environment

Before or after you install the tools, you can define *environment variables* that set certain code generation tool parameters you normally use. An environment variable is a system symbol that you define and assign to a string. When you use environment variables, default values are set, making each individual invocation of the tools simpler because these parameters are automatically specified. When you invoke a tool, you can use command line options to override many of the defaults that are set with environment variables.

The code generation tools use environment variables for finding or obtaining certain types of information. By default, the installation program sets up these environment variables:

```
set PATH=C:\tool_dir;%PATH%
set A_DIR=C:\tool_dir
set C_DIR=C:\tool_dir
set DOS4GVM=VirtualMemory:ON
set DOS4G=
```

If you choose not to have the environment variables set up automatically, you can modify your autoexec.bat file to include the SET commands above.

The remainder of this section describes these environment variables and other variables that you can define.

Identifying the directory that contains the executable files (PATH statement)

You must include the *tool_dir* directory in your PATH statement. This allows you to specify the assembler and compiler tools without specifying the name of the directory that contains the executable files.

Modify your autoexec.bat file to change the path information, and add the following to the end of the PATH statement:

;C:\*tool_dir*

Identifying alternate directories for the assembler (A_DIR)

By default, the assembler searches for copy/include files or macro libraries in the current directory and then in directories named by the –i (name alternate directories) option. Use the A_DIR environment variable to define additional search paths. The format of the command for assigning the environment variable is as follows:

set A_DIR=*pathname1*[*;pathname 2 . . .*]

You can separate the *pathnames* with a semicolon or a blank.

Once you set A_DIR, you can use the .copy, .include, or .mlib directive in assembly source without specifying path information.

For more information on the –i option, see the *TMS320C3x/C4x Assembly Language Tools User's Guide* or the *TMS320C3x/C4x Optimizing C Compiler User's Guide*.

Identifying alternate directories for the compiler (C_DIR)

By default, the compiler searches the current directory for #include files and object libraries such as the runtime-support and C I/O libraries. Use the C_DIR environment variable to define additional search paths. The format of the command for assigning the environment variable is as follows:

set C_DIR=*pathname1*[*;pathname 2 . . .*]

You can separate the *pathnames* with a semicolon or a blank.

Once you set C_DIR, you can use the #include directive in your C source code without specifying path information.

Setting default shell options (C_OPTION)

When using the shell program (cl30), you might find it useful to set default options for the compiler, assembler, and linker using the C_OPTION environment variable. After reading the command line options and input filenames, the shell reads the contents of the C_OPTION environment variable. Notice that options defined with C_OPTION do not override the options specified on the command line.

Setting up default options with the C_OPTION environment variable is especially useful when you want to run consecutive times with the same set of options and/or input files. The options and/or input filenames that you define with C_OPTION are used every time you run the shell.

The syntax for the C_OPTION environment variable is:

set C_OPTION=*option1*[*option2 . . .*]

Options specified in the environment variable are specified in the same way and have the same meaning as they do on the command line. For more information about compiler and assembler options, see the *TMS320C3x/C4x Optimizing C Compiler User's Guide* and the *TMS320C3x/C4x Assembly Language Tools User's Guide*.

For example, if you want to always run quietly (the –q option), enable C source interlisting (the –s option), and link (the –z option), set up the C_OPTION environment variable as follows:

```
set C_OPTION=–qs –z
```

In this example, each time you run the shell program, it runs the linker. Any options following –z on the command line or in C_OPTION are passed to the linker. This enables you to use the C_OPTION environment variable to specify default compiler and linker options and then specify additional compiler and linker options on the shell command line. If you have set –z in the environment variable and want to compile only, use the –c option of the shell. These additional examples assume C_OPTION is set as shown above:

```
cl30  *.c               ; compiles and links
cl30  –c *.c            ; only compiles
cl30  *.c –z c30.cmd    ; compiles and links using a
                        ; command file
cl30  –c *.c –z c30.cmd ; only compiles (–c overrides –z)
```

For more information on linker options, see the *Linker Options* section in the *TMS320C3x/C4x Assembly Language Tools User's Guide.*

Using virtual memory (DOS4GVM)

Virtual memory management (VMM) allows protected-mode programs to use more RAM than your computer actually has available. The DOS4GVM environment variable controls VMM. You can set the DOS4GVM environment variable using the following format:

set DOS4GVM=[*option***:***value*] [*option***:***value*] ...

You must use a colon before the value for each option; the DOS command shell does not accept an equal sign in place of the colon. Unless you specify otherwise for your system, these options are not case sensitive.

DOS4GVM options take effect only when VMM is enabled, which causes the default values to be used for all options. To enable VMM, enter:

```
set DOS4GVM=VirtualMemory:ON
```

The DOS4GVM options are described below:

DeleteSwapFile **DeleteSwapFile:**{**ON**|**OFF**}

By default, a new swap file is created each time your code runs, which slows down program startup. If you run your code over and over again and you can spare the disk space, you can set DeleteSwapFile to OFF so that an existing swap file is reused. If your code spawns another program that uses extended memory, do not set DeleteSwapFile.

PhysMax **PhysMax:***n*{**K**|**M**|**G**}

PhysMax specifies the maximum amount of physical memory (RAM) managed by VMM. The default is all available memory up to 64 Mbytes. This setting minimizes disk swapping so that large programs run as fast as possible. However, the default setting might actually slow down small programs on machines with a lot of memory because more memory is managed than is actually needed.

You might want to restrict the amount of physical memory VMM uses for the following reasons:

❑ Your code is small, and you know its maximum memory requirement. You can speed up startup by telling VMM not to manage everything.

❑ You need to spawn another program that uses extended memory, and you need to leave enough memory available for the other program.

PhysMin **PhysMin:***n*{**K**|**M**|**G**}

PhysMin specifies the minimum amount of physical memory (RAM) managed by VMM. The default is 1024 Kbytes. Set PhysMin to the minimum hardware requirement necessary for running your code. If your code requires a 4-Mbyte machine, set PhysMin to 4 Mbytes. If your code is small and can run in 512 Kbytes, set PhysMin to 512 Kbytes.

SwapFileName **SwapFileName:**[*path*][*filename*]

SwapFileName specifies the filename of the swap file. The default filename is DOS4GVM.SWP, and the file is placed in the directory where the executable file resides. Specify the complete path name if you want to keep the swap file in another directory.

SwapInc **SwapInc:***n*{**K**|**M**|**G**}

SwapInc specifies the size by which the swap file grows. The default size is 4096 Kbytes. As your program runs, the swap file increases by 4 Mbytes when the file needs to grow. A smaller size causes the swap file to take less time to increase but to increase more frequently. A larger size causes the swap file to take longer to increase but to increase less frequently.

To have a static swap file rather than a dynamic swap file, set SwapInc to 0 and set SwapMin to the static size you want.

SwapMin **SwapMin:***n*{**K**|**M**|**G**}

SwapMin specifies the minimum or initial size of the swap file. The default is 0 bytes. If you want VMM to create a full-size swap file at startup time, set SwapMin to the full size of the swap file and set SwapInc to 0.

VirtualSize **VirtualSize:***n*{**K**|**M**|**G**}

VirtualSize specifies the size of the virtual memory space. The default is 16384 Kbytes. Set VirtualSize to a larger size if your program uses more than 16 Mbytes of code and data, but do not set it to more than twice the size of your program's memory requirement.

You can change the defaults in two ways:

1) Specify parameter values as arguments to the DOS4GVM environment variable, as shown in the example below. Note that you must have at least 8192 Kbytes of free memory to use this example:

```
set DOS4GVM=deleteswapfile:ON physmax:8192K swapfilename:c:\swap.tmp
```

2) Create a configuration file with the filetype extension .VMC, and call that file as an argument to the DOS4GVM environment variable, as shown below:

   ```
   set DOS4GVM=@NEW4G.VMC
   ```

   A .VMC file contains VMM parameters and settings, as shown in the example below. You can include comments. Comments on lines by themselves must be preceded by an exclamation point (!). Comments that follow option settings must be preceded by white space. Do not insert blank lines; processing stops at the first blank line.

   ```
   !Sample .VMC file
   !This file shows the default parameter values
   physmin:512K        At least 512 bytes of RAM required
   physmax:4M          Uses no more than 4MB of RAM
   virtualsize:16M     Swap file + allocated memory is 16MB
   ```

See Appendix A, *Troubleshooting DOS Systems*, for information on problems and solutions when using DOS/4GW.

Specifying a temporary file directory (TMP)

The shell program creates intermediate files as it processes your program. For example, the parser phase of the shell creates a temporary file used as input by the code generator phase. By default, the shell puts intermediate files in the current directory. However, you can name a specific directory for temporary files by using the TMP environment variable.

This feature allows the use of a RAM disk or other file systems. It also allows source files to be compiled from a remote directory without writing any files into the directory where the source resides; this is useful for protected directories.

To set the TMP environment variable, use this syntax:

set TMP=*pathname*

For example, to set up a directory named temp for intermediate files on your hard drive, enter:

```
set TMP=C:\temp
```

1.4 Performance Considerations

You may notice a speed degradation when you use the code generation tools. Much of this speed degradation is due to the switch rate from protected to real mode necessitated by DOS calls. Higher-speed processors and later-generation processors in the 80386, 80486, and Pentium series minimize the time needed for this switch.

Virtual-memory management (VMM) may also degrade system performance. It is recommended that VMM be enabled only for programs that cannot be built with VMM disabled.

If you encounter error messages when you use the code generation tools on a PC with DOS, run PMINFO to determine the configuration of your system before you contact technical support. For more information about PMINFO, see Appendix A, *Troubleshooting DOS Systems*.

1.5 Where to Go From Here

Your code generation tools are now installed. At this point, you should do the following:

- ❑ Go to Chapter 5, *Getting Started With the Code Generation Tools*. This chapter provides you with an overview of how to invoke and use the assembler, linker, and compiler.
- ❑ Read Chapter 6, *Release Notes*, to understand the new features included in the 5.00 release of the code generation tools.
- ❑ Use Appendix A, *Troubleshooting DOS Systems*, as necessary. This appendix lists kernel and DOS/4G error messages and explains how you can resolve the messages.

Chapter 2

Setting Up the Tools on a PC Running Windows NT or Windows 95

This chapter helps you install release 5.00 of the TMS320C3x/C4x code generation tools and set up your code-development environment on a PC running Windows NT™ or Windows 95. These tools include an optimizing C compiler and a full set of assembly language tools for developing and manipulating assembly language and object (executable) code.

The C compiler tools include:

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

The assembly language tools are composed of the following:

- ❑ Assembler
- ❑ Archiver
- ❑ Linker
- ❑ Absolute lister
- ❑ Cross-reference lister
- ❑ Hex conversion utility

Topic **Page**

2.1 System Requirements

To install the TMS320C3x/C4x code generation tools on a PC, you need the following:

- ❑ 32-bit x86 or Pentium™-based PC running
  - Windows NT 3.51 (or higher)
  - Windows 95
- ❑ 4–16 Mbytes of free memory. 16 Mbytes is recommended to ensure optimum performance when compiling large C functions
- ❑ 10 Mbytes of disk space for the executable files and libraries
- ❑ CD-ROM drive

Note: Memory Needed

The code generation tools, when installed on a PC, require at least 4 Mbytes of memory, but you can expect some performance problems when using only 4 Mbytes. (16 Mbytes is recommended.) You may want to free as much memory as possible before installing the tools, especially if you have less than 16 Mbytes.

2.2 Installing the Tools

The code generation tools package is shipped on CD-ROM. To install the tools on a PC running Windows NT or Windows 95, follow these steps:

1) Insert the TMS320C3x/C4x Software Toolkit CD-ROM into your CD-ROM drive.
2) Start Windows NT or Windows 95.
3) From the File menu (Windows NT 3.51) or the Start menu (Windows NT 4.0 or Windows 95), select Run.
4) In the dialog box, enter the following command (replace d with the name of your CD-ROM drive):

   ```
   d:\setup.exe
   ```
5) Click on OK.
6) Follow the on-screen instructions.

2.3 Setting Up the Environment

Before or after you install the tools, you can define *environment variables* that set certain code generation tool parameters you normally use. An environment variable is a system symbol that you define and assign to a string. When you use environment variables, default values are set, making each individual invocation of the tools simpler because these parameters are automatically specified. When you invoke a tool, you can use command line options to override many of the defaults that are set with environment variables.

The code generation tools use environment variables for finding or obtaining certain types of information. By default, the installation program sets up these environment variables:

```
set PATH=C:\tool_dir;%PATH%
set A_DIR=C:\tool_dir
set C_DIR=C:\tool_dir
```

The remainder of this section describes these environment variables and other variables, and discusses alternative ways that variables can be defined.

Setting environment variables under Windows NT

Under Windows NT, the environment variables are set up in the registry under:

```
HKEY_CURRENT_USER\ENVIRONMENT
```

If you choose not to have the environment variables automatically set up in the registry, it is recommended that you set up the environment variables in the System applet of the Control Panel.

To set up the environment variables in the System applet of the Control Panel, simply open the System applet and for each environment variable listed above, enter the name of the Variable and its associated Value, then select Set. In the System applet, you can make the environment variables available to all users or you can define them for specific individuals.

Setting environment variables under Windows 95

Under Windows 95, the environment variables are set up in your autoexec.bat file.

If you choose not to have the environment variables set up automatically, you can modify your autoexec.bat file manually to include the SET commands above.

Identifying the directory that contains the executable files (PATH statement)

You must include the *tool_dir* directory in your PATH statement. This allows you to specify the assembler and compiler tools without specifying the name of the directory that contains the executable files.

- ❑ If you modify your autoexec.bat file to change the path information, add the following to the end of the PATH statement:

  ;C:\*tool_dir*

- ❑ If you set the PATH statement from the command line, use this format:

  set PATH=C:\*tool_dir***;%PATH%**

  The addition of **;%PATH%** ensures that this PATH statement does not undo the PATH statements in any other batch files (including the autoexec.bat file).

Identifying alternate directories for the assembler (A_DIR)

By default, the assembler searches for copy/include files or macro libraries in the current directory and then in directories named by the –i (name alternate directories) option. Use the A_DIR environment variable to define additional search paths. The format of the command for assigning the environment variable is as follows:

set A_DIR=*pathname1*[*; pathname 2 . . .*]

You can separate the *pathnames* with a semicolon or a blank.

Once you set A_DIR, you can use the .copy, .include, or .mlib directive in assembly source without specifying path information.

For more information on the –i option, see the *TMS320C3x/C4x Assembly Language Tools User’s Guide* or the *TMS320C3x/C4x Optimizing C Compiler User’s Guide*.

Identifying alternate directories for the compiler (C_DIR)

By default, the compiler searches the current directory for #include files and object libraries such as the runtime-support and C I/O libraries. Use the C_DIR environment variable to define additional search paths. The format of the command for assigning the environment variable is as follows:

set C_DIR=*pathname1*[*; pathname 2 . . .*]

You can separate the *pathnames* with a semicolon or a blank.

Once you set C_DIR, you can use the #include directive in your C source code without specifying path information.

Setting default shell options (C_OPTION)

When using the shell program (cl30), you might find it useful to set default options for the compiler, assembler, and linker using the C_OPTION environment variable. After reading the command line options and input filenames, the shell reads the contents of the C_OPTION environment variable. Notice that options defined with C_OPTION do not override the options specified on the command line.

Setting up default options with the C_OPTION environment variable is especially useful when you want to run consecutive times with the same set of options and/or input files. The options and/or input filenames that you define with C_OPTION are used every time you run the shell.

The syntax for the C_OPTION environment variable is:

set C_OPTION=*option1*[*option2 . . .*]

Options specified in the environment variable are specified in the same way and have the same meaning as they do on the command line. For more information about compiler and assembler options, see the *TMS320C3x/C4x Optimizing C Compiler User's Guide* and the *TMS320C3x/C4x Assembly Language Tools User's Guide*.

For example, if you want to always run quietly (the –q option), enable C source interlisting (the –s option), and link (the –z option), set up the C_OPTION environment variable as follows:

```
set C_OPTION=–qs –z
```

In this example, each time you run the shell program, it runs the linker. Any options following –z on the command line or in C_OPTION are passed to the linker. This enables you to use the C_OPTION environment variable to specify default compiler and linker options and then specify additional compiler and linker options on the shell command line. If you have set –z in the environment variable and want to compile only, use the –c option of the shell. These additional examples assume C_OPTION is set as shown above:

```
cl16  *.c                ; compiles and links
cl16  –c *.c             ; only compiles
cl16  *.c –z c30.cmd     ; compiles and links using a
                         ; command file
cl16  –c *.c –z c30.cmd  ; only compiles (–c overrides –z)
```

For more information on linker options, see the *Linker Options* section in the *TMS320C3x/C4x Assembly Language Tools User's Guide.*

Specifying a temporary file directory (TMP)

The shell program creates intermediate files as it processes your program. For example, the parser phase of the shell creates a temporary file used as input by the code generator phase. By default, the shell puts intermediate files in the current directory. However, you can name a specific directory for temporary files by using the TMP environment variable.

This feature allows use of a RAM disk or other file systems. It also allows source files to be compiled from a remote directory without writing any files into the directory where the source resides; this is useful for protected directories.

To set the TMP environment variable, use this syntax:

set TMP=*pathname*

For example, to set up a directory named temp for intermediate files on your hard drive, enter:

```
set TMP=C:\temp
```

2.4 Where to Go From Here

Your code generation tools are now installed. At this point, you should do the following:

❑ Go to Chapter 5, *Getting Started With the Code Generation Tools*. This chapter provides you with an overview of how to invoke and use the assembler, linker, and compiler.

❑ Read Chapter 6, *Release Notes*, to understand the new features included in the 5.00 release of the code generation tools.

Chapter 3

Setting Up the Tools on a SPARCstation

This chapter helps you install release 5.00 of the TMS320C3x/C4x code generation tools and set up your code-development environment on a SPARCstation running SunOS™ versions 4.1.x (or higher). These tools include an optimizing C compiler and a full set of assembly language tools for developing and manipulating assembly language and object (executable) code.

The C compiler tools include:

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

The assembly language tools are composed of the following:

- ❏ Assembler
- ❏ Archiver
- ❏ Linker
- ❏ Absolute lister
- ❏ Cross-reference lister
- ❏ Hex conversion utility

This release is dynamically linked to take advantage of shared libraries.

Topic **Page**

3.1 System Requirements

To install the TMS320C3x/C4x code generation tools on a SPARCstation, you need the following items:

- ❑ SPARCstation or compatible system with SPARCstation 2 class or higher performance
- ❑ 4 Mbytes of disk space for the software tools
- ❑ OpenWindows™ version 3.0 (or higher) running under SunOS version 4.1.x (or higher). If you are using SunOS 5.x (also known as Solaris 2.x), you must have the Binary Compatibility Package (BCP) installed; if you do not, get your system administrator’s help.
- ❑ CD-ROM drive
- ❑ Root privileges to mount and unmount the CD-ROM if you have SunOS 4.1.x, SunOS 5.0, or SunOS 5.1

3.2 Mounting the CD-ROM and Installing the Tools

To install the software tools, you must mount the CD-ROM, copy the files to your system, and unmount the CD-ROM.

Note:

If you are running SunOS 4.1.x, 5.0, or 5.1, you *must* have root privileges to mount or unmount the CD-ROM. If you do not, get help from your system administrator.

Mounting the CD-ROM

The code generation tools package is shipped on CD-ROM. The steps to mount the CD-ROM vary according to your operating system version:

❑ If you have a SunOS 4.1.x, as root, load the CD-ROM into the drive and enter the following from a command shell:

```
# mount –rt hsfs /dev/sr0 /cdrom
# exit
% cd /cdrom/sparc
```

❑ If you have SunOS 5.0 or 5.1, as root, load the CD-ROM into the drive and enter the following from a command shell:

```
# mount –rF hsfs /dev/sr0 /cdrom
# exit
% cd /cdrom/cdrom0/sparc
```

❑ If you have SunOS 5.2 or higher:

- If the CD-ROM drive is already attached, load the CD-ROM into the drive. Enter:

  ```
  % cd /cdrom/cdrom0/sparc
  ```

- If the CD-ROM drive is not attached, you must shut down your system to the PROM level (at the OK prompt) and attach the CD-ROM drive. As root, enter:

  ```
  # boot –r
  ```

  Log on, load the CD-ROM into the drive, and enter:

  ```
  % cd /cdrom/cdrom0/sparc
  ```

Installing the tools

Be sure you are not logged on as root. To install the software tools, follow these steps:

1) If you do not already have a tools directory, create one. Enter:

   ```
   mkdir tool_dir
   ```

   Replace *tool_dir* with your own directory name, including the path information, to install the tools.

2) Copy the files to your directory:

   ```
   cp –r * tool_dir
   ```

Unmounting the CD-ROM

You must unmount the CD-ROM after copying the files.

❑ If you have a SunOS 4.1.x, SunOS 5.0, or SunOS 5.1, as root, enter:

```
# cd
# umount /cdrom
# eject /dev/sr0
# exit
```

❑ If you have SunOS 5.2 or higher, enter:

```
% cd
% eject
```

3.3 Setting Up the Environment

Before or after you install the tools, you can define *environment variables* that set certain code generation tool parameters you normally use. An environment variable is a system symbol that you define and assign to a string. When you use environment variables, default values are set, making each individual invocation of the tools simpler because these parameters are automatically specified. When you invoke a tool, you can use command line options to override many of the defaults that are set with environment variables.

To set up the environment, enter these commands. Be sure you are not logged on as root.

- For C shells:

```
setenv C_DIR ”tool_dir”
setenv A_DIR ”tool_dir”
set path=(tool_dir $path)
```

- For Bourne or Korn shells:

```
C_DIR=tool_dir
A_DIR=tool_dir
PATH=tool_dir:$PATH
```

You can move these commands into your .login or .cshrc file (for C shells) or .profile file (for Bourne or Korn shells) to avoid entering the commands each time you invoke a new shell.

The remainder of this section describes these environment variables and other variables that you can define.

Identifying the directory that contains the executable files (path statement)

You must include the *tool_dir* directory in your path statement. This allows you to specify the assembler and compiler tools without specifying the name of the directory that contains the executable files.

- If you modify your .cshrc file (for C shells) or .profile file (for Bourne or Korn shells) to change the path information, add the following to the end of the path statement:

  tool_dir

- If you set the path statement from the command line, use this format:
  - For C shells:

    set path=(*tool_dir* **$path)**
  - For Bourne or Korn shells:

    PATH=*tool_dir* **$PATH**

  The addition of **$path**/**$PATH** ensures that this path statement does not undo the path statements in the .cshrc or .profile file.

Identifying alternate directories for the assembler (A_DIR)

By default, the assembler searches for copy/include files or macro libraries in the current directory and then in directories named by the –i (name alternate directories) option. Use the A_DIR environment variable to define additional search paths. The format of the command for assigning the environment variable is as follows:

- ❑ For C shells:

  setenv A_DIR ***"****pathname1*;*pathname2* … ***"***

- ❑ For Bourne or Korn shells:

  A_DIR=*"**pathname1*;*pathname2* … ***"***
  export A_DIR

You can separate the *pathnames* with a semicolon or a blank.

Once you set A_DIR, you can use the .copy, .include, or .mlib directive in assembly source without specifying path information.

For more information on the –i option, see the *TMS320C3x/C4x Assembly Language Tools User's Guide* or the *TMS320C3x/C4x Optimizing C Compiler User's Guide*.

Identifying alternate directories for the compiler (C_DIR)

By default, the compiler searches the current directory for #include files and object libraries such as the runtime-support and C I/O libraries. Use the C_DIR environment variable to define additional search paths. The format of the command for assigning the environment variable is as follows:

- ❑ For C shells:

  setenv C_DIR ***"****pathname1*;*pathname2* … ***"***

- ❑ For Bourne or Korn shells:

  C_DIR=*"**pathname1*;*pathname2* … ***"***
  export C_DIR

You can separate the *pathnames* with a semicolon or a blank.

Once you set C_DIR, you can use the #include directive in your C source code without specifying path information.

Setting default shell options (C_OPTION)

When using the shell program (cl30), you might find it useful to set default options for the compiler, assembler, and linker using the C_OPTION environment variable. After reading the command line options and input filenames, the shell reads the contents of the C_OPTION environment variable. Notice that options defined with C_OPTION do not override the options specified on the command line.

Setting up default options with the C_OPTION environment variable is especially useful when you want to run consecutive times with the same set of options and/or input files. The options and/or input filenames that you define with C_OPTION are used every time you run the shell.

The syntax for the C_OPTION environment variable is:

- For C shells:

  setenv C_OPTION *"option1 option2 … "*

- For Bourne or Korn shells:

  C_OPTION=*"option1 option2 … "*
  export C_OPTION

Options specified in the environment variable are specified in the same way and have the same meaning as they do on the command line. For more information about compiler and assembler options, see the *TMS320C3x/C4x Optimizing C Compiler User's Guide* and the *TMS320C3x/C4x Assembly Language Tools User's Guide*.

For example, if you want to always run quietly (the –q option), enable C source interlisting (the –s option), and link (the –z option), set up the C_OPTION environment variable as follows:

```
setenv C_OPTION "–qs –z"     ; for C shells
C_OPTION="–qs –z"            ; for Bourne or Korn shells
export C_OPTION
```

In this example, each time you run the shell program, it runs the linker. Any options following –z on the command line or in C_OPTION are passed to the linker. This enables you to use the C_OPTION environment variable to specify default compiler and linker options and then specify additional compiler and linker options on the shell command line. If you have set –z in the environment variable and want to compile only, use the –c option of the shell. These additional examples assume C_OPTION is set as shown above:

```
cl30  *.c                ; compiles and links
cl30  –c *.c             ; only compiles
cl30  *.c –z c30.cmd     ; compiles and links using a
                         ; command file
cl30  –c *.c –z c30.cmd  ; only compiles (–c overrides –z)
```

For more information on linker options, see the *Linker Options* section in the *TMS320C3x/C4x Assembly Language Tools User’s Guide.*

Specifying a temporary file directory (TMP)

The shell program creates intermediate files as it processes your program. For example, the parser phase of the shell creates a temporary file used as input by the code generator phase. By default, the shell puts intermediate files in the current directory. However, you can name a specific directory for temporary files by using the TMP environment variable.

This feature allows use of a RAM disk or other file systems. It also allows source files to be compiled from a remote directory without writing any files into the directory where the source resides; this is useful for protected directories.

To set the TMP environment variable, use this syntax:

- ❑ For C shells:

  ```
  setenv TMP ”/temp”
  ```

- ❑ For Bourne or Korn shells:

  ```
  TMP=”/temp”
  export TMP
  ```

3.4 Where to Go From Here

Your code generation tools are now installed. At this point, you should do the following:

- ❑ Go to Chapter 5, *Getting Started With the Code Generation Tools*. This chapter provides you with an overview of how to invoke and use the assembler, linker, and compiler.
- ❑ Read Chapter 6, *Release Notes*, to understand the new features included in the 5.00 release of the code generation tools.

Chapter 4

Setting Up the Tools on an HP Workstation

This chapter helps you install release 5.00 of the TMS320C3x/C4x code generation tools and set up your code-development environment on an HP 9000 Series 700™ PA-RISC™ computer with HP-UX™ 9.0x. These tools include an optimizing C compiler and a full set of assembly language tools for developing and manipulating assembly language and object (executable) code.

The C compiler tools include:

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

The assembly language tools are composed of the following:

- ❑ Assembler
- ❑ Archiver
- ❑ Linker
- ❑ Absolute lister
- ❑ Cross-reference lister
- ❑ Hex conversion utility

Topic **Page**

4.1 System Requirements

To install the TMS320C3x/C4x code generation tools on an HP workstation, you need the following items:

❑ HP 9000 Series 700 PA-RISC computer

❑ 4 Mbytes of disk space for the software tools

❑ HP-UX 9.0x operating system

❑ CD-ROM drive

❑ Root privileges to mount and unmount the CD-ROM

4.2 Mounting the CD-ROM and Installing the Tools

To install the software tools, you must mount the CD-ROM, copy the files to your system, and unmount the CD-ROM.

Note:

You *must* have root privileges to mount or unmount the CD-ROM. If you do not, get help from your system administrator.

Mounting the CD-ROM

The code generation tools package is shipped on CD-ROM. As root, you can mount the CD-ROM using the UNIX™ mount command or the SAM (System Administration Manager):

❑ To use the UNIX mount command, enter:

```
# mount –rt cdfs /dev/dsk/your_cdrom_device /cdrom
# exit
```

Make the hp directory on the CD-ROM the current directory. For example, if the CD-ROM is mounted at /cdrom, enter:

```
% cd /cdrom/hp
```

❑ To use SAM to mount the CD-ROM, see *System Administration Tasks*, the HP documentation about SAM, for instructions.

Installing the tools

Be sure you are not logged on as root. To install the software tools, follow these steps:

1) If you do not already have a tools directory, create one. Enter:

```
mkdir tool_dir
```

Replace *tool_dir* with your own directory name, including the path information, to install the tools.

2) Copy the files to your directory:

```
cp –r * tool_dir
```

Unmounting the CD-ROM

You must unmount the CD-ROM after copying the files. As root, enter:

```
# cd
# umount /cdrom
# exit
```

4.3 Setting Up the Environment

Before or after you install the tools, you can define *environment variables* that set certain code generation tool parameters you normally use. An environment variable is a system symbol that you define and assign to a string. When you use environment variables, default values are set, making each individual invocation of the tools simpler because these parameters are automatically specified. When you invoke a tool, you can use command line options to override many of the defaults that are set with environment variables.

To set up the environment, enter these commands. Be sure you are not logged on as root.

❑ For C shells:

```
setenv C_DIR "tool_dir"
setenv A_DIR "tool_dir"
set path=(tool_dir $path)
```

❑ For Bourne or Korn shells:

```
C_DIR=tool_dir
A_DIR=tool_dir
PATH=tool_dir:$PATH
```

You can move these commands into your .login or .cshrc file (for C shells) or .profile file (for Bourne or Korn shells) to avoid entering the commands each time you invoke a new shell.

The remainder of this section describes these environment variables and other variables that you can define.

Identifying the directory that contains the executable files (path statement)

You must include the *tool_dir* directory in your path statement. This allows you to specify the assembler and compiler tools without specifying the name of the directory that contains the executable files.

❑ If you modify your .cshrc file (for C shells) or .profile file (for Bourne or Korn shells) to change the path information, add the following to the end of the path statement:

  tool_dir

❑ If you set the path statement from the command line, use this format:

  ■ For C shells:

   set path=(*tool_dir* **$path)**

  ■ For Bourne or Korn shells:

   PATH=*tool_dir* **$PATH**

  The addition of **$path**/**$PATH** ensures that this path statement does not undo the path statements in the .cshrc or .profile file.

Identifying alternate directories for the assembler (A_DIR)

By default, the assembler searches for copy/include files or macro libraries in the current directory and then in directories named by the –i (name alternate directories) option. Use the A_DIR environment variable to define additional search paths. The format of the command for assigning the environment variable is as follows:

- ❑ For C shells:

  setenv A_DIR ***"****pathname1*;*pathname2* … ***"***

- ❑ For Bourne or Korn shells:

  A_DIR=*"**pathname1*;*pathname2* … ***"***
  export A_DIR

You can separate the *pathnames* with a semicolon or a blank.

Once you set A_DIR, you can use the .copy, .include, or .mlib directive in assembly source without specifying path information.

For more information on the –i option, see the *TMS320C3x/C4x Assembly Language Tools User's Guide* or the *TMS320C3x/C4x Optimizing C Compiler User's Guide*.

Identifying alternate directories for the compiler (C_DIR)

By default, the compiler searches the current directory for #include files and object libraries such as the runtime-support and C I/O libraries. Use the C_DIR environment variable to define additional search paths. The format of the command for assigning the environment variable is as follows:

- ❑ For C shells:

  setenv C_DIR ***"****pathname1*;*pathname2* … ***"***

- ❑ For Bourne or Korn shells:

  C_DIR=*"**pathname1*;*pathname2* … ***"***
  export C_DIR

You can separate the *pathnames* with a semicolon or a blank.

Once you set C_DIR, you can use the #include directive in your C source code without specifying path information.

Setting default shell options (C_OPTION)

When using the shell program (cl30), you might find it useful to set default options for the compiler, assembler, and linker using the C_OPTION environment variable. After reading the command line options and input filenames, the shell reads the contents of the C_OPTION environment variable. Notice that options defined with C_OPTION do not override the options specified on the command line.

Setting up default options with the C_OPTION environment variable is especially useful when you want to run consecutive times with the same set of options and/or input files. The options and/or input filenames that you define with C_OPTION are used every time you run the shell.

The syntax for the C_OPTION environment variable is:

- ❑ For C shells:

  setenv C_OPTION *”option1 option2 … ”*

- ❑ For Bourne or Korn shells:

  C_OPTION=*”option1 option2 … ”*
  export C_OPTION

Options specified in the environment variable are specified in the same way and have the same meaning as they do on the command line. For more information about compiler and assembler options, see the *TMS320C3x/C4x Optimizing C Compiler User’s Guide* and the *TMS320C3x/C4x Assembly Language Tools User’s Guide*.

For example, if you want to always run quietly (the –q option), enable C source interlisting (the –s option), and link (the –z option), set up the C_OPTION environment variable as follows:

```
setenv C_OPTION ”–qs –z”    ; for C shells
C_OPTION=”–qs –z”           ; for Bourne or Korn shells
export C_OPTION
```

In this example, each time you run the shell program, it runs the linker. Any options following –z on the command line or in C_OPTION are passed to the linker. This enables you to use the C_OPTION environment variable to specify default compiler and linker options and then specify additional compiler and linker options on the shell command line. If you have set –z in the environment variable and want to compile only, use the –c option of the shell. These additional examples assume C_OPTION is set as shown above:

```
cl30  *.c                ; compiles and links
cl30  –c *.c             ; only compiles
cl30  *.c –z c30.cmd     ; compiles and links using a
                         ; command file
cl30  –c *.c –z c30.cmd  ; only compiles (–c overrides –z)
```

For more information on linker options, see the *Linker Options* section in the *TMS320C3x/C4x Assembly Language Tools User's Guide.*

Specifying a temporary file directory (TMP)

The shell program creates intermediate files as it processes your program. For example, the parser phase of the shell creates a temporary file used as input by the code generator phase. By default, the shell puts intermediate files in the current directory. However, you can name a specific directory for temporary files by using the TMP environment variable.

This feature allows use of a RAM disk or other file systems. It also allows source files to be compiled from a remote directory without writing any files into the directory where the source resides; this is useful for protected directories.

To set the TMP environment variable, use this syntax:

- ❑ For C shells:

  ```
  setenv TMP ”/temp”
  ```

- ❑ For Bourne or Korn shells:

  ```
  TMP=”/temp”
  export TMP
  ```

4.4 Where to Go From Here

Your code generation tools are now installed. At this point, you should do the following:

- ❑ Go to Chapter 5, *Getting Started With the Code Generation Tools*. This chapter provides you with an overview of how to invoke and use the assembler, linker, and compiler.
- ❑ Read Chapter 6, *Release Notes*, to understand the new features included in the 5.00 release of the code generation tools.

Chapter 5

Getting Started With the Code Generation Tools

This chapter helps you start using the assembler, linker, and compiler tools by providing basic startup information. For more information about invoking and using these tools, see the *TMS320C3x/C4x Assembly Language Tools User's Guide* and the *TMS320C3x/C4x Optimizing C Compiler User's Guide*.

Topic **Page**

5.1 Getting Started With the Assembler and Linker

This section provides a quick walkthrough of the assembler and linker so that you can get started without reading the entire *TMS320C3x/C4x Assembly Language Tools User’s Guide*. These examples show the most common methods for invoking the assembler and linker.

Create two short source files to use for the walkthrough; call them file1.asm and file2.asm. (See Example 5–1 and Example 5–2.)

Example 5–1. file1.asm

```
          .file ”file1.asm”
          .ref addvec
          .global __stack
          .global start

__stack   .usect ”.stack”, 0

          .text
stack_a   .word __stack
vector    .float 10.0, 20.0, 30.0, 40.0
vector_a  .word vector
start:
          ldp @stack_a
          ldi @stack_a, sp
          ldi @vector_a, ar0
          call addvec
          bu $
```

Example 5–2. file2.asm

```
     .file ”file2.asm”
     .def addvec

     .text

addvec:
     ldf 0,r0
     rpts 3
     addf *ar0++,r0
     rets
```

1) Enter the following command to assemble file1.asm:

   `asm30 file1` ⏎

   The **asm30** command invokes the assembler. The input source file is file1.asm. (If the input file extension is .asm, you do not have to specify the extension; the assembler uses .asm as the default.)

   This example creates an object file called file1.obj. The assembler creates an object file only if there are no errors. You can specify a name for the object file, but if you do not, the assembler uses the input filename with an extension of .obj.

Note: The –v30 Option is the Assembler Default

The **asm30** command invokes the TMS320C3x/C4x assembler. By default, the assembler generates code for the ’C30, as if the –v30 option had been used.

Use the –v31 option to generate code for the ’C31. Use the –v32 option to generate code for the ’C32. Use the –v40 option to generate code for the ’C40. Use the –v44 option to generate code for the ’C44.

2) Now enter the following command to assemble file2.asm:

   `asm30 file2.asm –l` ⏎

   This time, the assembler creates an object file called file2.obj. The –l (lowercase L) option tells the assembler to create a listing file; the listing file for this example is called file2.lst. It is not necessary to create a listing file, but it gives you information and assures you that the assembly has resulted in the desired object code. The listing file for this example is shown in Example 5–3.

Example 5–3. file2.lst, the Listing File Created by asm30 file2.asm –l

```
TMS320C3x/4x COFF Assembler Version 5.00
Fri Dec 20 11:01:34 1996
Copyright (c) 1987–1996 Texas Instruments Incorporated

file2.asm PAGE 1

1                              .file ”file2.asm”
2                              .def addvec
3 00000000                     .text
4 00000000 07608000  addvec:   ldf 0,r0
5 00000001 13fb0003            rpts 3
6 00000002 01c02001            addf *ar0++,r0
7 00000003 78800000            rets

No Errors, No Warnings
```

3) Now enter the following command to link file1.obj and file2.obj:

   `lnk30 file1 file2 –m lnker2.map –o prog.out` ⏎

The lnk30 command invokes the linker. The input object files are file1.obj and file2.obj. (If the input file extension is .obj, you do not have to specify the extension; the linker uses .obj as the default.) The linker combines file1.obj and file2.obj to create an executable object module called prog.out. The –o option supplies the name of the output module. Example 5–4 shows the map file resulting from this operation. The map file is produced only if you use the –m option.

Example 5–4. Output Map File, lnker2.map

```
*****************************************************
TMS320C3x/4x COFF Linker   Version 5.00
*****************************************************
Fri Dec 20 11:03:11 1996

OUTPUT FILE NAME: <prog.out>
ENTRY POINT SYMBOL: 0

SECTION ALLOCATION MAP

output                                    attributes/
section   page   origin       length       input sections
--------  ----   ----------   ----------   ----------------
.text     0      00000000     0000000f
                   00000000     0000000b   file1.obj (.text)
                   0000000b     00000004   file2.obj (.text)

.data     0      00000000     00000000     UNINITIALIZED
                   00000000     00000000   file1.obj (.data)
                   00000000     00000000   file2.obj (.data)

.bss      0      00000000     00000000     UNINITIALIZED
                   00000000     00000000   file1.obj (.bss)
                   00000000     00000000   file2.obj (.bss)

.stack    0      0000000f     00000400     UNINITIALIZED
                   0000000f     00000000   file1.obj (.stack)

GLOBAL SYMBOLS

address name                  address name
-------- ----                 -------- ----
00000000 .bss                 00000000 edata
00000000 .data                00000000 .data
00000000 .text                00000000 end
00000400 __STACK_SIZE         00000000 .bss
0000000f __stack              00000000 .text
0000000b addvec               00000006 start
00000000 edata                0000000b addvec
00000000 end                  0000000f etext
0000000f etext                0000000f __stack
00000006 start                00000400 __STACK_SIZE

[10 symbols]
```

5.2 Getting Started With the C Compiler

The TMS320C3x/C4x C compiler consists of two passes: the first pass parses the code, and the second pass produces a single assembly language source file that must be assembled and linked. You can specify an optional optimization pass after the first pass. The simplest way to compile, assemble, and link a C program is to use the compiler shell program with the –z option. This section provides a quick walkthrough so that you can get started without reading the entire *TMS320C3x/C4x Optimizing C Compiler User's Guide*.

1) Create a sample file called function.c that contains the code shown in Example 5–5.

Example 5–5. Sample C File for the C Compiler Walkthrough, function.c

```
/************************************************************/
/*                      function.c                         */
/*             (Sample file for walkthrough)               */
/************************************************************/

int abs_func(int i)
{
        int temp = 1;
        if (temp < 0) temp *= –1;
        return (temp);
}

void main(void)
{
        int x = –3;
        x = abs_func(x);
}
```

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

   `cl30 –o function`

   The –o option invokes the optimizer at the default level. The shell program prints the following information as it compiles the program:

Example 5–6. Diagnostic Messages Produced by the Compiler

```
[function]
TMS320C3x/4x ANSI C Compiler   Version 5.00
Copyright (c) 1987–1997  Texas Instruments Incorporated
   ”function.c”   ==> abs_func
   ”function.c”   ==> main
TMS320C3x/4x ANSI C Optimizer   Version 5.00
Copyright (c) 1987–1997  Texas Instruments Incorporated
   ”function.c”   ==> abs_func
   ”function.c”   ==> main
TMS320C3x/4x C Code Generator  Version 5.00
Copyright (c) 1987–1997  Texas Instruments Incorporated
   ”function.c”   ==> abs_func
   ”function.c”   ==> main
TMS320C3x/4x COFF Assembler    Version 5.00
Copyright (c) 1987–1997  Texas Instruments Incorporated
 PASS 1
 PASS 2

 No Errors,  No Warnings
```

The shell program runs the two compiler passes, the optimizer, and the assembler as follows:

ac30	$\rightarrow$	C parser
opt30	$\rightarrow$	Optimizer
cg30	$\rightarrow$	Code generator
asm30	$\rightarrow$	Assembler

By default, the shell deletes the assembly language file from the current directory after the file is assembled. If you want to inspect the assembly language output, use the –k option to retain the assembly language file:

```
cl30 –o –k function
```

3) Also by default, the shell creates a COFF object file as output; however, if you use the –z option, the output is an *executable* object module. The following examples show two ways of creating an executable object module:

   a) The example in step 2 creates an object file called function.obj. To create an executable object module, run the linker separately by invoking lnk30 as in the following example:

```
lnk30 –c function.obj c30.cmd –o function.out –l rts30.lib
```

The –c linker option tells the linker to observe the C language linking conventions. The linker command file, lnk.cmd, is shipped with the code generation tools. The –o option names the output module, function.out; if you do not use the –o option, the linker names the output module a.out. The –l option names the runtime-support library. You must have a runtime-support library before you can create an executable object module; the prebuilt runtime-support library, rts.lib, is included with the code generation tools.

b) In this example, use the –z option, which tells the shell program to run the linker. The –z option is followed by linker options.

```
cl30 –o function.c –z c30.cmd –o function.out –l rts30.lib
```

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

ac30	→	C parser
opt30	→	Optimizer
cg30	→	Code generator
asm30	→	Assembler
lnk30	→	Linker

For more information on linker commands, see the *Linker Description* chapter of the *TMS320C3x/C4x Assembly Language Tools User's Guide* and the *Linking C Code* chapter of the *TMS320C3x/C4x Optimizing C Compiler User's Guide.*

4) The TMS320C3x/C4x compiler package also includes an *interlist utility*. This program interlists the C source statements as comments in the assembly language compiler output, allowing you to inspect the assembly language generated for each line of C. To run the interlist utility, invoke the shell program with the –ss option. For example:

```
cl30 –ss function –z c30.cmd –o function.out
```

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

Chapter 6

Release Notes

This chapter contains documentation of tools and features that are new or have been changed since the last release. It details all enhancements made to the TMS320C3x/C4x floating-point DSP assembly language tools and optimizing C compiler.

Topic **Page**

6.1 Release Enhancements

This section lists the release enhancements for version 5.00 of the TMS320C3x/C4x floating-point DSP code generation tools. Each enhancement includes a reference to the manual in which it is further detailed. The following abbreviations are used:

ALT	See the *TMS320C3x/C4x Assembly Language Tools User's Guide* (literature number: SPRU035C)
COMP	See the *TMS320C3x/C4x Optimizing C Compiler User's Guide* (literature number: SPRU034G)

- ❑ The OS/2 operating system is not supported in this release
- ❑ The following command line options have been added (COMP, Section 2.1.3):

–ad*name*	Predefines the constant *name* for the assembler
–au*name*	Undefines the constant *name* for the assembler
–ml	Runtime support assembly calls use far calls
–mp	Performs speed optimizations at the cost of increased code size
–ms	Assumes all memory is accessible when optimizing
–mtc	Generates an additional header for every C function compiled, allowing it to be used with the Tartan LAJ function calling method
–os	Interlists optimizer comments into the compiler's assembly language output
–ss	Invokes the interlist utility, which interlists C source statements into the compiler's assembly language output

- ❑ The –mx command line option has been removed
- ❑ Six new runtime support libraries are included: (COMP, Section 2.3)
  rts30g.lib, rts30gr.lib, rts30r.lib
  rts40g.lib, rts40gr.lib, rts40r.lib
- ❑ New intrinsics have been added:
  fast_ftoi(), ansi_ftoi(), fast_imult(), fast_invf() (COMP, Section 2.8)
- ❑ Long doubles are now represented in extended–precision 40-bit format (COMP, Section 3.2.1)
- ❑ The compiler supports far calls (COMP, Section 3.7)

- ❑ The compiler includes enhanced support for filling the three delay slots generated for branches (COMP, Section 3.8)
- ❑ The big memory model places constants in a .const section (COMP, Section 4.2.5)
- ❑ The register calling convention has been modified. All registers are now assigned by the compiler. (COMP, Section 4.3)
- ❑ There are three runtime-support arithmetic functions specifically for extended-precision arithmetic: MPY_LD (multiply), DIV_LD (divide), and INV_LD (inverse/reciprocals). These functions support the 40-bit long double data type. (COMP, Section 4.7)
- ❑ The compiler uses the MPYI instruction more effectively for integer multiplication (COMP, Section 4.7)
- ❑ When the –mp option is specified, the SQR inline function is used for squaring, which reduces cycle time (COMP, Section 4.7)
- ❑ The runtime-support library includes C I/O support (COMP, Appendix B)
- ❑ The following version symbols have been added to the assembler: (ALT, Section 3.9.3)
  .C3x, .C30, .C31, .C32, .C4x, .C40, .C44
- ❑ The assembler supports the .regalias directive, which allows registers named F*n* to be aliases for floating-point versions of R*n* data registers (ALT, Chapter 4)
- ❑ The archiver (ar30) now accepts command files (ALT, Section 7.2)
- ❑ The archiver includes the –u option. This option, used in conjunction with the –r option, causes the archiver to replace specified members in the library only if they have a more recent modification date. (ALT, Section 7.2)
- ❑ The absolute lister (abs30), a debugging tool that allows you to create a listing of absolute addresses of object code, has been added to the product (ALT, Chapter 9)
- ❑ The hex converter includes the –m1, –m2, and –m3 options to support the Motorola S1, S2, and S3 formats, respectively. (ALT, Section 10.11.3)
- ❑ COFF2 is the default format generated by the compiler, assembler, and linker. This version of COFF supports subsections, conditional linking, and long names. (ALT, Chapter 2 and Appendix A)
- ❑ Benchmark improvements average 10% over version 4.70

6.2 Useful Tips

- The –mf compiler option ensures that pointer access to an external variable never uses direct addressing. If you do not use the –mf option, the compiler may use direct addressing if a local variable pointer is assigned the address of the external variable. Use of the option will not make all addressing to all external variables indirect.

- When the compiler generates code for the ’C4x, it assumes that the SET COND bit in the ST register is set to 0. The compiler will not work correctly if the SET COND is set to 1.

- When you use the –x option with asm30 or the –ax option with cl30, a COPY section with cross reference information is included in the COFF file. Although the COPY section is not supposed to be loaded, TI loaders with core debugger versions older than 3.33 will attempt to load the section. You can use the –mv option on the debugger command line to find the debugger core version.

  The ’C3x simulator version 2.20 and the ’C4x simulator version 1.30/1.31 loaders have this problem. The following loaders do not have the problem:

  - ’C4x XDS510/XDS510WS debugger version 2.40
  - ’C3x XDS510/XDS510WS debugger version 5.00

  A work-around to this problem is to specify in the SECTIONS directive of the linker command file that the .xref section be allocated at a specific memory location that will not disturb the program. The other option is not to use the –x option if you intend to use one of the affected loaders.

- DOS restricts the length of the command line to a total of 128 characters. The 128-character restriction also exists for the command lines that cl30 creates for calling the other tools (ac30, opt30, cg30, etc.). When these command lines are generated, the entire path is included for the executable (for example c:\tools\rel460\ac30.exe). This means that more options can be included if you run the tools from the directory where they exist or if you shorten the length of the path to the executable files.

  Another solution is to use the –@ shell option. This option causes the compiler to read shell options and commands from a command file.

- You cannot nest GROUP (group output sections) and UNION directives in a linker command file. You can, however, group input sections within a UNION directive (A GROUP directive groups output sections). This accomplishes the same thing since the linker continuously allocates output sections. For example:

```
MEMORY
{
     EXT0    : org = 0x001000,   len = 0x0800
     RAM0    : org = 0x809800,   len = 0x0400
}
SECTIONS
{
       UNION  run = RAM0
       {
            .text1:  load = EXT0
            {
                  file1.obj(.text)
                  file2.obj(.text)
            }
            .text2:  load = EXT0
            {
                  file3.obj(.text)
                  file4.obj(.text)
            }
       }

}
```

- If you have multiple sections that need to start at the same run address and they require alignment by different amounts, you can use overlays. This is one way to simulate what would happen if you could nest GROUP directives in UNION directives. The one thing to be aware of is that some loaders may use the page information to place code into different banks of memory. All of the TI loaders (EVM, emulator, and simulator) do not use the page information in the COFF file.

```
MEMORY
{
     PAGE 0 : RAM0    : org = 0x001000,   len = 0x0800
     PAGE 1 : RAM0    : org = 0x809800,   len = 0x0400
     PAGE 2 : RAM0    : org = 0x809800,   len = 0x0400
}

SECTIONS
{
     .text1 :   load = RAM0   PAGE 0,   run = RAM0    PAGE 1   align 128
     .text2 :   load = RAM0   PAGE 0,   run = RAM0    PAGE 1   align 64
     .text3 :   load = RAM0   PAGE 0,   run = RAM0    PAGE 2   align 512
     .text4 :   load = RAM0   PAGE 0,   run = RAM0    PAGE 2   align 8
}
```

- An alternative to using mk30 to build a library is to use use cl30 to compile all of the .c and .h files and then use ar30 to archive all of the .obj files to an object archive:

  SPARC:

  ```
  cd ~/float/rts
  ar30 –x ~/float/rts.src
  cl30 –o3 –ol0 –op0 –i. *
  ar30 –a rts.lib *.obj
  ```

  DOS:

  ```
  cd c:\float\rts
  ar30 –x c:\float\rts.src
  cl30 –o3 –ol0 –op0 –i. *.*
  ar30 –a rts.lib *.obj
  ```

- The branch instructions (BR, BRD, CALL, RPTB, RPTBD, LAJ, B*cond*, B*cond*AF, B*cond*AT, B*cond*D, CALL*cond*, DB*cond*, DB*cond*D, LAJ*cond*, LAT*cond*, RETI*cond*, RETI*cond*D, RETS*cond*, etc.) have either 16- or 24-bit displacement fields (+/– 32K or +/– 8M words, respectively). In most cases, the compiler uses the conditional branches with 16-bit offsets. However, there are several ways that you can specify 24–bit calls:

  - Use a function pointer to call the function. This will use the register addressing mode rather than the PC-displacement addressing mode.
  - Use the far modifier when defining the function. For example,

    ```
    extern far int function(parameters);
    ```

    tells the compiler to use the 24–bit version of the call.
  - Use the –ml option. This option tells the compiler to use 24–bit far calls when calling internal RTS functions such as MPY_I30.

- Using the .sect directive to define a user definable initialized section in C code does not always work correctly. This method is often used when you want to give a portion of code different run and load addresses. The compiler may generate .text information, then it may generate .const or .cinit information, and then go back to generating more .text information. This means that a portion of the code that was intended for the user section may be placed in the .text section.

  The .label directive can be used with any initialized section like .text, .data, or other user definable initialized sections (.sect directive) to place a label at the load address. It does not make sense to have a run address for the .cinit section since the data is copied to .bss at boot time or load time and then never referenced again. Uninitialized sections like .bss, .stack or .sysmem only have runtime addresses. Therefore, it does not make sense to have a separate load address.

A general method of placing C code at a separate load address is illustrated in the following code segments. The example consists of four C files, move.c, move1.c, move2.c, and exit.c, and an assembly language file, boot.asm. The C file exit.c and the assembly language file boot.asm are not reproduced here as they are included in the runtime support library. The linker command file move.cmd controls the linking process. The five files are compiled, assembled, and linked with the following command line:

```
cl30 –g move.c move1.c move2.c exit.c boot.asm –z move.cmd
```

The file move.c contains the function main_run() that we want to relocate at runtime. The asm statement is used to embed TMS320C3x/C4x assembly language statements. The .sects statement create named sections called .load and .end. The .label statement creates a special label that refers to the load address of the labels __load_addr and __load_end that mark the beginning and ending of the code to be relocated at run time.

```
********************** move.c **********************

asm("                  .sect   \".load\"       ");
asm("                  .global __load_addr     ");
asm("                  .global __run_addr      ");
asm("                  .label  __load_addr     ");
asm("__run_addr        .text                   ");

extern int test(int in);

int main_run(int in)
{
        int out = test(in);
        return out;
}

asm("                  .sect   \".end\"        ");
asm("                  .global __load_end      ");
asm("                  .label  __load_end      ");
```

The file move1.c contains the C code that moves the function main_run() from its load address to its run address.

```
********************** move1.c **********************
extern int _load_addr;
extern int _load_end;
extern int _run_addr;
extern int main_run(int in);
main()
{
        int (*ptr_main)(int n) = main_run;
        int *load_addr  = &_load_addr;
        int *load_end   = &_load_end;
        int *run_addr   = &_run_addr;
        int len         = load_end – load_addr;
        int i;
        for(i=0; i<len; i++)
               *run_addr++ = *load_addr++;
        i = (*ptr_main)(100);
}
```

The file move2.c contains a function call to the function test(). Note that the code that relocates the function must run before the function can be called.

```
********************** move2.c **********************
int test(int in)
{
         return in;
}
```

The file move.cmd contains a linker command file for the preceding example. The most important part of this command file for the purpose of the example is the SECTIONS directive that allocates move.obj and move2.obj to load into external memory and run in RAM block 1. Note that the linker command file allocates the space, but it is the file move1.c that contains the code that actually moves functions main_run() and test() from their load addresses to their run addresses.

```
******************************** move.cmd *************************************
-c                                           /* USE C ROM MODEL                 */
-stack 0x100                                 /* STACK                           */
-heap  0x100                                 /* HEAP                            */
-lrts30.lib                                  /* RUN-TIME SUPPORT LIBRARY        */
-m c3x.map                                   /* GENERATE MAP FILE               */
-o c3x.out                                   /* NAME OUTPUT FILE                */
-x                                           /* REREAD LIBRARIES                */
-w                                           /* WARN                            */

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
   ROM:    org = 0x0      len = 0x1000     /* INTERNAL 4K ROM                 */
   EXT0:   org = 0x1000   len = 0x7ff000   /* EXTERNAL MEMORY                 */
   XBUS:   org = 0x800000 len = 0x2000     /* EXPANSION BUS                   */
   IOBUS:  org = 0x804000 len = 0x2000     /* I/O BUS                         */
   RAM0:   org = 0x809800 len = 0x400      /* RAM BLOCK 0                     */
   RAM1:   org = 0x809c00 len = 0x400      /* RAM BLOCK 1                     */
   EXT1:   org = 0x80a000 len = 0x1000     /* EXTERNAL MEMORY                 */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
   .boot:   > EXT0
   {
        boot.obj (.text)
        move1.obj (.text)
        exit.obj (.text)
   }
   .text:   load = EXT0, run = RAM1
   {
        move.obj (.load)
        * (.text)
        move.obj (.end)
   }
   .cinit:  > RAM1                   /* INITIALIZATION TABLES                 */
   .const:  > RAM1                   /* CONSTANTS                             */
   .stack:  > RAM1                   /* SYSTEM STACK                          */
   .sysmem: > RAM1                   /* DYNAMIC MEMORY - DELETE IF NOT USED */
   .bss:    > RAM1, block 0x10000    /* VARIABLES                             */
}
```

Appendix A

Troubleshooting DOS Systems

DOS/4GW is a memory manager that is embedded into the TMS320C3x/C4x code generation tools, so you may occasionally see DOS/4GW error messages while you are using the tools. The executable files for DOS/4GW are not shipped as such, nor is any documentation provided on this tool, except for the list of error messages.

Section A.2, *Kernel Error Messages*, and Section A.3, *DOS/4G Error Messages*, are excerpted from the *DOS/4GW User's Manual* (reproduced here with the permission of Tenberry Software, Inc.) Included are lists of error messages with descriptions of the circumstances in which the error is most likely to occur and suggestions for remedying the problem. (Portions of the excerpt have been modified to provide you with specific information about using TI tools.)

Topic **Page**

A.1 Troubleshooting in the Protected-Mode Environment

Getting 32-bit programs to execute properly under DOS can be frustrating. Your computer's configuration and memory management can cause problems that may be difficult to find because many programs are interacting.

This list of error messages is reproduced here because they may occur when executing any tools, since all of the tools have been assembled along with the DOS/4GW memory extender. When reading this material, keep these considerations in mind:

❑ When an *Action* directs you to technical support, determine the configuration of your system by using the **PMINFO** (on page A-3) programs before contacting technical support:

  To contact technical support, call the following telephone number:

DSP Hotline	**(281) 274–2320**

❑ Some error messages are not included in this section because they are rarely seen when using DOS/4GW with the TMS320C3x/C4x tools. Also, many of the messages that *are* documented here are seldom seen when using DOS/4GW with the TMS320C3x/C4x tools. Nevertheless, you may find this text to be useful in debugging your programs.

Should you encounter any error message not listed here, or should problems persist, contact technical support as directed above.

The PMINFO.EXE program

Purpose: Run PMINFO.EXE to determine the performance of protected/real-mode switching and extended memory.

Notes: The time-based measurements made by PMINFO may vary slightly from run to run.

If this error message appears:

DOS/16M error: [17] system software does not follow VCPI or DPMI specifications

check for a statement in your CONFIG.SYS containing **NOEMS**. If such a statement exists, remove it and reboot your computer.

If the computer is not equipped with extended memory or if none is available for DOS/4GW, the extended-memory measurements will not display.

Other DOS/4GW error messages are in Section A.3, *DOS/4G Error Messages*.

Example: The following example shows the output of the PMINFO program on an 80486 AT-compatible machine running at 33 MHz.

```
–================================= PMINFO ========================================–

           Protected Mode and Extended Memory Performance Measurement –– 4.45
                   Copyright (c) Tenberry Software, Inc. 1987 – 1995

      DOS memory   Extended memory   CPU performance equivalent to 33.0 MHz 80486
      ----------   ---------------
             640             17854   K bytes configured (according to BIOS).
             640             31744   K bytes physically present (SETUP).
             550             17585   K bytes available for DOS/16M programs.
      21.6 (0.0)        19.1 (0.5)   MB/sec word transfer rate (wait states).
      35.4 (0.5)        34.4 (0.5)   MB/sec 32–bit transfer rate (wait states).

      Overall cpu and memory performance (non–floating point) for typical
      DOS programs is 7.78 ± 0.62 times an 8MHz IBM PC/AT.

      Protected/Real switch rate = 18078/sec (55 µsec/switch, 33 up + 21 down),
      DOS/16M switch mode 11 (VCPI).
```

PMINFO provides the information shown in Table A–1.

Table A–1. PMINFO Fields

Measurement	Purpose
CPU performance	Shows the CPU processor equivalent and the speed of the CPU (in MHz).
According to BIOS	Shows the configured memory in DOS and extended memory as provided by the BIOS (interrupts 12h and 15h, function 88h).
SETUP	Shows the configuration obtained directly from the CMOS RAM as set by the computer's setup program. It is displayed only if the numbers are different from those in the BIOS line. They are different if the BIOS has reserved memory for itself or if another program has allocated memory and is intercepting the BIOS configuration requests to report less memory available than is physically configured.
DOS/16M programs	If displayed, shows the low and high addresses available to DOS/4GW in extended memory.
Transfer rates	PMINFO tries to determine the memory architecture. Some architectures perform well under some circumstances and poorly under others; PMINFO shows the best and worst cases. The architectures detected are cache, interleaved, page-mode (or static column), and direct. Measurements are made by using 32-bit accesses and are reported as the number of megabytes per second that can be transferred. The number of wait states is reported in parentheses. The wait states can be a fractional number, like 0.5, if there is a wait state on writes but not on reads. Memory bandwidth (that is, how fast the CPU can access memory) accounts for 60% to 70% of the performance for typical programs (those that are not heavily dependent on floating-point math).
Overall CPU and memory performance	Shows a performance metric developed by Tenberry Software, Inc. (formerly known as Rational Systems, Inc.), indicating the expected throughput for the computer relative to a standard 8-MHz IBM PC/AT (disk accesses and floating-point operations are both excluded).
Protected/real switch rate	Shows the speed with which the computer can switch between real and protected modes, both as the maximum number of round-trip switches that can occur per second, and as the time for a single round-trip switch, broken into the real-to-protected (up) and protected-to-real (down) components.

A.2 Kernel Error Messages

This section describes error messages from the DOS/16M kernel embedded in the TMS320C3x/C4x code generation tools. Kernel error messages can occur because of severe resource shortages, corruption of the executable file, corruption of memory, operating system incompatibilities, or internal errors. All of these messages are quite rare.

DOS/16M protected mode available only with 386 or 486

Description DOS/4G did not detect the presence of a 386, 486, or Pentium-based processor. You may see this error message even if you are using a 386 PC or later.

Action If you are running the tools on a 386 (or later) PC, rerun the program. If you are running the tools on a 286 PC, reinstall and run the tools on a 386 PC or later.

0: involuntary switch to real mode

Description The computer was in protected mode but switched to real mode without going through DOS/16M. This error most often occurs because of an unrecoverable stack segment exception (stack overflow) but can also occur if the Global Descriptor Table or Interrupt Descriptor Table is corrupted.

Action Restart your computer. If the problem persists, contact technical support.

2: not a DOS/16M executable <filename>

Description DOS4G.EXE or a bound DOS/4G application has probably been corrupted in some way.

Action Recopy the file from the source media.

6: not enough memory to load program

Description There is not enough memory to load DOS/4G.

Action Make more memory available and try again.

8: cannot open file <filename>

Description The DOS/16M loader cannot load DOS/4G, probably because DOS has run out of file units.

Action Set a larger FILES= entry in the CONFIG.SYS file, reboot, and try again.

9: cannot allocate tstack

Description	There is not enough memory to load DOS/4G.
Action	Make more memory available and try again.

10: cannot allocate memory for GDT

Description	There is not enough memory to load DOS/4G.
Action	Make more memory available and try again.

11: no passup stack selectors – GDT too small

Description	There is an internal error in DOS/4G or an incompatibility with other software.
Action	Contact technical support.

12: no control program selectors – GDT too small

Description	There is an internal error in DOS/4G or an incompatibility with other software.
Action	Contact technical support.

13: cannot allocate transfer buffer

Description	There is not enough memory to load DOS/4G.
Action	Make more memory available and try again.

14: premature EOF

Description	DOS4G.EXE or a bound DOS/4G application has probably been corrupted.
Action	Recopy the file from the source media.

15: protected mode available only with 386 or 486

Description	DOS/4G requires an 80386 (or later) CPU. It cannot run on an 80286 (or earlier) CPU.
Action	Reinstall and run the tools on a 386 (or later) PC.

17: system software does not follow VCPI or DPMI specifications

Description	Some memory-resident program has put your 386 or 486 CPU into Virtual 8086 mode. This is done to provide special memory services to DOS programs, such as EMS simulation (EMS interface without EMS hardware) or high memory. In this mode, it is not possible to switch into protected mode unless the resident software follows a standard that DOS/16M supports (DPMI, VCPI, and XMS are the most common).
Action	Contact the vendor of your memory-management software.

22: cannot free memory

Description	Memory was probably corrupted during execution of your program.
Action	Make more memory available and try again.

23: no memory for VCPI page table

Description	There is not enough memory to load DOS/4G.
Action	Make more memory available and try again.

24: VCPI page table address incorrect

Description	This is an internal error.
Action	Contact technical support.

25: cannot initialize VCPI

Description	An incompatibility with other software was detected. DOS/16M has detected that VCPI is present, but VCPI returns an error when DOS/16M tries to initialize the interface.
Action	Find the other software that uses VCPI and disable it (stop its execution).

28: memory error, avail loop

Description	Memory was probably corrupted during execution of your program. Using an invalid or stale alias selector may cause this error. Incorrect manipulation of segment descriptors may also cause it.
Action	Rerun the program and/or restart your computer.

29: memory error, out of range

Description	Memory was probably corrupted during execution of your program. Writing through an invalid or stale alias selector may cause this error.
Action	Check your source code for references to variables that are not declared or are no longer in scope.

32: DPMI host error (possibly insufficient memory)
33: DPMI host error (need 64K XMS)
34: DPMI host error (cannot lock stack)

Description	Memory under DPMI is probably insufficient.
Action	Under Windows, make more physical memory available by eliminating or reducing any RAM drives or disk caches. You can also edit DEFAULT.PIF so that at least 64K bytes of XMS memory is available to non-Windows programs. Under OS/2, increase the DPMI_MEMORY_LIMIT in the DOS box settings.

35: general protection fault

Description	An internal error in DOS/4G was probably detected. Faults generated by your program should cause error 2001 instead.
Action	Contact technical support.

38: cannot use extended memory: HIMEM.SYS not version 2

Description	An incompatibility with an old version of HIMEM.SYS was detected.
Action	Upgrade to a more recent copy of DOS or upgrade your DOS memory extender.

40: not enough available extended memory (XMIN)

Description	An incompatibility with your memory manager or its configuration was detected.
Action	Configure the memory manager to provide more extended memory or change memory managers.

A.3 DOS/4G Error Messages

DOS/4G errors are more common than kernel errors when using DOS/4G or DOS/4GW with the TMS320C3x/C4x code generation tools. They are usually related to an unknown path name, corrupt files, or memory problems. Memory problems can include inadequate memory, poor configuration, or corrupted memory.

1000 ”can’t hook interrupts”

Description	A DPMI host has prevented DOS/4G from loading.
Action	Contact technical support.

1001 ”error in interrupt chain”

Description	A DOS/4G internal error was detected.
Action	Contact technical support.

1003 ”can’t lock extender kernel in memory”

Description	DOS/4G couldn’t lock the kernel in physical memory, probably because of a memory shortage.
Action	Free some memory for the DOS/4G application.

1005 ”not enough memory for dispatcher data”

Description	There is not enough memory for DOS/4G to manage user-installed interrupt handlers properly.
Action	Free some memory for the DOS/4G application.

1007 ”can’t find file <program> to load”

Description	DOS/4G could not open the specified program. The file probably does not exist. It is possible that DOS ran out of file handles or that a network or similar utility has prohibited read access to the program.
Action	Make sure that the filename was spelled correctly.

1008 ”can’t load executable format for file <filename> [<error code>]”

Description	DOS/4G did not recognize the specified file as a valid executable file. DOS/4G can load linear executables (LE and LX) and EXPs (BW).
Action	Recopy the file from the source media.

3301 ”unhandled EMPTYFWD, GATE16, or unknown relocation”
3302 ”unhandled ALIAS16 reference to unaliased object”
3304 ”unhandled or unknown relocation”

Description	If your program was built for another platform that supports the LINEXE format, it may contain a construct that DOS/4G does not currently support, such as a call gate. One of these messages may also appear if your program has a problem mixing 16- and 32-bit code. A linker error is another likely cause.
Action	Check for viruses and reinstall the tools from the source media. If the problem persists, contact technical support.

Appendix B

Tables of Peripheral Registers, Structure-Member Names, and Bit-Field Names

The TMS2320C3x peripheral control library provides C data structures for manipulating the TMS320C3x peripherals. The tables in this appendix list the data structure member names that are used to access each of the peripheral registers and bit fields through C peripheral pointers. For a detailed explanation of the register and bit-field descriptions, refer to the *TMS320C3x User’s Guide*.

This appendix provides an update to the parallel runtime-support table.

The first entry for each register shows how to access that register as an integer. The remaining entries show how to access the register’s bit fields individually. Each table is followed by an example.

Table B–1. Bus Control Registers

Register	Assignment	Bit Field	Member name
STRB0 Bus Control	Integer	——	–>strb0_gcontrol
	Bit-field	STRB Switch	–>strb0_gcontrol_bit.strbsw
		STRB Config	–>strb0_gcontrol_bit.strbcnfg
		Sign Ext/Zero Fill	–>strb0_gcontrol_bit.signext
		Physical Memory Width	–>strb0_gcontrol_bit.memwidth
		Data Size	–>strb0_gcontrol_bit.datasize
		BNKCMP	–>strb0_gcontrol_bit.bnkcmp
		WTCNT	–>strb0_gcontrol_bit.wtcnt
		SWW	–>strb0_gcontrol_bit.sww
		HIZ	–>strb0_gcontrol_bit.hiz
		NOHOLD	–>strb0_gcontrol_bit.nohold
		HOLDST	–>strb0_gcontrol_bit.holdst
STRB1 Bus Control	Integer	——	–>strb1_gcontrol
	Bit-field	Sign Ext/Zero Fill	–>strb1_gcontrol_bit.signext
		Physical Memory Width	–>strb1_gcontrol_bit.memwidth
		Data Size	–>strb1_gcontrol_bit.datasize
		BNKCMP	–>strb1_gcontrol_bit.bnkcmp
		WTCNT	–>strb1_gcontrol_bit.wtcnt
		SWW	–>strb1_gcontrol_bit.sww
IOSTRB Bus Control	Integer	——	–>strb1_gcontrol
	Bit-field	WTCNT	–>strb1_gcontrol_bit.wtcnt
		SWW	–>iostrb_gcontrol_bit.sww

Example B–1. Bus Control

```
#include <bus32.h>
BUS_REG *bus_ptr=BUS_ADDR;    /* Define pointer to bus peripheral  */
bus_ptr–>iostrb_gcontrol = 0;          /* zero wait states on IOSTRB bus    */
```

Table B–2. DMA Control Registers

Register	Assignment	Bit FIeld	Member Name
DMA0 Global Control	Integer	——	–>gcontrol
	Bit-field	PRIORITY MODE	–>gcontrol_bit.pri_mode
		DMA PRI	–>gcontrol_bit.dma_pri
		TCINT	–>gcontrol_bit.tcint
		TC	–>gcontrol_bit.tc
		SYNC	–>gcontrol_bit.sync
		DECDST	–>gcontrol_bit.decdst
		INCDST	–>gcontrol_bit.incdst
		DECSRC	–>gcontrol_bit.decsrc
		INCSRC	–>gcontrol_bit.incsrc
		STAT	–>gcontrol_bit.stat
		START	–>gcontrol_bit.start
DMA1 Global Control	Integer	——	–>gcontrol
	Bit-field	TCINT	–>gcontrol_bit.tcint
		TC	–>gcontrol_bit.tc
		SYNC	–>gcontrol_bit.sync
		DECDST	–>gcontrol_bit.decdst
		INCDST	–>gcontrol_bit.incdst
		DECSRC	–>gcontrol_bit.decsrc
		INCSRC	–>gcontrol_bit.incsrc
		STAT	–>gcontrol_bit.stat
		START	–>gcontrol_bit.start
DMAx Source Address	Integer	——	–>source
DMAx Destination Address	Integer	——	–>destination
DMAx Transfer Counter	Integer	——	–>transfer_counter

Example B–2. DMA Control

```
#include <dma32.h>
DMA_REG *dma0=DMA_ADDR(0);        /* Define pointer to DMA0   */
dma0–>gcontrol_bit.start=STOP;             /* Stop DMA by setting start bits */
```

Appendix C

Glossary

A

ANSI: American National Standards Institute. An organization that establishes standards voluntarily followed by industries.

asm30: The name of the command that invokes the assembler for the TMS320C3x/C4x.

assembler: A software program that creates a machine-language program from a source file that contains assembly language instructions, directives, and macro directives. The assembler substitutes absolute operation codes for symbolic operation codes, and absolute or relocatable addresses for symbolic addresses.

B

.bss: One of the default COFF sections. You can use the .bss directive to reserve a specified amount of space in the memory map that can later be used for storing data. The .bss section is uninitialized.

C

C compiler: A program that translates C source statements into assembly language source statements.

cl30: The name of the compiler shell program for the TMS320C3x/C4x. (Note that the second character in the shell name is a lowercase L.)

common object file format (COFF): A binary object file format that promotes modular programming by supporting the concept of *sections.*

D

DOS/4G: The base version for DOS/4GW. You may occasionally see this term in an error message. If so, refer to Appendix A, *Troubleshooting DOS Systems*, for the appropriate action.

DOS/4GW: A memory extender that is bound with the MS-DOS version of the TMS320C3x/C4x tools. *The executable DOS/4GW file is not shipped separately but is embedded within the other executables*. Error messages from DOS/4GW are included in Appendix A, *Troubleshooting DOS Systems*, to assist you in debugging. If you receive one of these error messages, contact technical support for assistance, and remember that the tools are shipped as object files with the memory extender embedded.

DOS/16M: The executable filename for a tool that is embedded in the TMS320C3x/C4x code generation tools. You may occasionally see this term in an error message. If so, refer to Appendix A, *Troubleshooting DOS Systems*, for the appropriate action.

E

environment variables: System symbols that you define and assign to a string. They are usually included in batch files (for example, in the AUTOEXEC.BAT file).

G

global: A kind of symbol that is either 1) defined in the current module and accessed in another, or 2) accessed in the current module but defined in another.

I

initialized section: A COFF section that contains executable code or initialized data. An initialized section can be built up with the .data, .text, or .sect directive.

interlist utility: A utility that inserts as comments your original C source statements into the assembly language output from the assembler. The C statements are inserted next to the equivalent assembly instructions.

L

linker: A software tool that combines object files to form an object module that can be allocated into TMS320C3x/C4x system memory and executed by the device.

listing file: An output file created by the assembler that lists source statements, their line numbers, and their effects on the section program counter (SPC).

lnk30: The name of the command that invokes the linker for the TMS320C3x/C4x.

M

map file: An output file, created by the linker, that shows the memory configuration, section composition, section allocation, and symbol definitions and the addresses at which the symbols were defined for your program.

O

optimization: Improvement in the execution speed of a program or in the reduction of the size of C programs.

P

pragma: Preprocessor directive that provides directions to the compiler about how to treat a particular statement.

protected-mode programs: 32-bit extended MS-DOS programs. These programs require an extended memory manager and run on 80386-, 80486-, and Pentium-based PCs only. Protected-mode programs can use all available RAM on the computer up to 64 Mbytes.

R

real mode: 16-bit native MS-DOS mode. This mode limits the available memory to 640K bytes. Calls to DOS may involve switching from protected to real mode. *DOS real-mode tools are no longer supported by the TMS320C3x/C4x code generation tools.*

S

section: A relocatable block of code or data that will ultimately occupy contiguous space in the TMS320C3x/C4x memory map.

static variable: A kind of variable whose scope is confined to a function or a program. The values of static variables are not discarded when the function or program is exited; their previous value is resumed when the function or program is re-entered.

string table: A table that stores symbol names that are longer than eight characters (symbol names of eight characters or longer cannot be stored in the symbol table; instead, they are stored in the string table). The name portion of the symbol's entry points to the location of the string in the string table.

subsection: A relocatable block of code or data that will ultimately occupy contiguous space in the TMS320C3x/C4x memory map. Subsections are smaller sections within larger sections. Subsections give you tighter control of the memory map.

swap file: The file where virtual memory(secondary memory) is allocated on the hard disk.

symbolic debugging: The ability of a software tool to retain symbolic information so that it can be used by a debugging tool such as a simulator or an emulator.

T

.text: One of the default COFF sections. The .text section is an initialized section that contains executable code. You can use the .text directive to assemble code into the .text section.

U

uninitialized section: A COFF section that reserves space in the memory map but that has no actual contents. These sections are built up with the .bss and .usect directives.

V

virtual memory: The ability of a program to use more memory than a computer actually has available as RAM. This is accomplished by using a swap file on disk to augment RAM. When RAM is not sufficient, part of the program is swapped out to a disk file until it is needed again. The combination of the swap file and available RAM is the virtual memory. The TMS370C16 tools use the DOS/4GW memory extender to provide virtual memory management (VMM). This memory extender is not provided as an executable file but is embedded in several of the tools shipped by TI. Contact technical support for more information.

Index