TMS320C8x Software Tools Getting Started Guide

Release 2.00

Literature Number: SPRU117C
Manufacturing Part Number: 2613611-9741 revision D
February 1997

Printed on Recycled Paper

IMPORTANT NOTICE

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

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

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

Copyright © 1997, Texas Instruments Incorporated

Preface

Read This First

About This Manual

This manual tells you how to install release 2.00 of the TMS320C8x software tools. It also does the following:

- Tells you how to set environment variables for parameters that you use often
- Provides you with a list of the media contents for your tools set, so you will know what information is associated with each file you have installed
- Gets you started using the compiler, linker, and assembler
- Details the enhancements included and tells you where to find further information

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 version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is an example of a command that you might enter:

  `ppasm pp_a –l`

- 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:

  setenv DISPLAY "*machinename***"**

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

  setenv D_OPTIONS "[*object filename*] [*debugger options*]**"**

Related Documentation From Texas Instruments

The following books describe the TMS320C8x 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.

TMS320C80 (MVP) C Source Debugger User's Guide (literature number SPRU107) describes the 'C8x master processor and parallel processor C source debuggers. This manual provides information about the features and operation of the debuggers and the parallel debug manager; it also includes basic information about C expressions and a description of progress and error messages.

TMS320C80 (MVP) Code Generation Tools User's Guide (literature number SPRU108) describes the 'C8x code generation tools. This manual provides information about the features and operation of the linker and the master processor (MP) and parallel processor (PP) C compilers and assemblers. It also includes a description of the common object file format (COFF) and shows you how to link MP and PP code.

TMS320C8x Master Processor User's Guide (literature number SPRU109) provides information about the master processor (MP) features, architecture, operation, and assembly language instruction set; it also includes sample applications that illustrate various MP operations.

TMS320C8x Multitasking Executive User's Guide (literature number SPRU112) provides information about the multitasking executive software features, operation, and interprocessor communications; it also includes a list of task error codes.

TMS320C8x Parallel Processor User's Guide (literature number SPRU110) provides information about the parallel processor (PP) features, architecture, operation, and assembly language instruction set; it also includes software applications and optimizations.

TMS320C8x System-Level Synopsis (literature number SPRU113) describes the 'C8x features, development environment, architecture, memory organization, and communication network (the crossbar).

TMS320C80 Transfer Controller User's Guide (literature number SPRU105) provides information about the transfer controller (TC) features, functional blocks, and operation; it also includes examples of block write operations for big- and little-endian modes.

TMS320C80 Video Controller User's Guide (literature number SPRU111) provides information about the video controller (VC) features, architecture, and operation; it also includes procedures and examples for programming the serial register transfer (SRT) controller and the frame timer registers.

TMS320C80 Digital Signal Processor Data Sheet (literature number SPRS023) describes the features of the TMS320C80 and provides pinouts, electrical specifications, and timings for the device.

TMS320C80 to TMS320C82 Software Compatibility User's Guide (literature number SPRU154) describes how to port software developed for one of these devices to the other. It also presents a set of software compatibility guidelines for developing software that will run on either device.

Modified Goertzel Algorithm in DTMF Detection Using the TMS320C80 Application Report (literature number SPRA066) describes the C-callable Goertzel dual-tone multi-frequency (DTMF) detection algorithm implementation on one of the TMS320C80's parallel processors.

Interfacing SDRAM to the TMS320C80 Application Report (literature number SPRA055) describes an interface between the 'C80 and the TMS626802-10 SDRAM(s). It illustrates the operation of the SDRAM interface and provides schematics for a baseline SDRAM interface to the TMS320C80.

Interfacing DRAM to the TMS320C80 Application Report (literature number SPRA056) describes an interface between the 'C80 and the TMS417400 DRAM(s). The report also describes the interface's connection to 4MB and 8MB SIMMs, a timing analysis for the proposed design, and a complete set of schematics.

TMS320C80 H.320 Software Library White Paper (literature number SPRY002) describes the TMS320C80's single-chip implementation of the H.320 videoconferencing standard.

TMS320C8x Software Development Board Installation Guide (literature number SPRU150B), included with the TMS320C8x SDB, provides information about how to install and use the SDB.

TMS320C8x Software Development Board Programmer's Guide (literature number SPRU178), included with the TMS320C8x SDB, provides descriptions of hardware functions, complete API references, theory of operation, and example code for the SDB.

TMS320C8x (DSP) — Fundamental Graphic Algorithms Application Book (literature number SPRA069) contains application notes that demonstrate several 'C8x application programs. These notes are as follows:

- ***Transform3 Command*** is an application presented for the Master Processor which demonstrates how two-dimensional graphic coordinate transformations can use the 'C8x floating-point unit efficiently. Performance estimation also is provided.
- ***Transform4 Command*** is an application presented for the Master Processor which demonstrates how three-dimensional graphic coordinate transformations can use the 'C8x floating-point unit efficiently. Performance estimation also is provided.
- ***Draw Colored Lines Command*** is an application presented for the 'C8x parallel processor which demonstrates how to generate colored lines efficiently. Performance estimation also is provided.
- ***Draw Colored Trapezoids Command*** is an application presented for the 'C8x parallel processor which demonstrates how to generate colored trapezoids efficiently. Performance estimation also is provided.
- ***Parallel Processor Integer and Floating-Point Math*** describes several C–callable 32–bit IEEE 754 standard floating–point math subroutines. Performance estimation also is provided.

Implementation of the Vector Maximum Search Benchmark on the TMS320C8x Parallel Processor Application Report (literature number SPRA087) uses the *Vector Maximum Search* benchmark to demonstrate the efficient performance of the TMS320C8x parallel processors. This manual describes a software implementation that uses the parallel processor's advanced assembly language features to implement this benchmark.

Acoustic Echo Cancellation — Algorithms and Implementation on the TMS320C8x Application Report (literature number SPRA063) describes the implementation of an integral N-tap digital acoustic echo canceller on the TMS320C8x parallel processor. The report presents a brief discussion of generic echo cancellation algorithms. The implementation considerations for a 512-tap (64-ms span) echo canceller on the TMS320C8x are described in detail, as well as the software logic and flow for each program module.

Viewing TMS320C8x Register Bit Fields and Memory-Mapped Registers in the HLL Debugger (TMS320 DSP Designer's Notebook, DNP# 69) describes a method for viewing 'C8x register bit fields and memory-mapped registers in the 'C8x HLL debugger.

Writing TMS320C8x PP Code Under the Multitasking Executive (TMS320 DSP Designer's Notebook, DNP# 73) provides useful guidelines for writing 'C8x Parallel Processor (PP) assembly language or C code that can run under the Multitasking Executive.

TMS320C82 Digital Signal Processor Data Sheet (literature number – Preliminary) describes the features of the TMS320C82 and provides pinouts, electrical specifications, and timings for the device.

Trademarks

320 Hotline On-line is a trademark of Texas Instruments Incorporated.

Microsoft, Windows, and Windows NT are registered trademarks of Microsoft Corporation.

OpenWindows, Solaris, and SunOS are trademarks of Sun Microsystems, Inc.

PC is a trademark of International Business Machines Corporation.

Pentium is a trademark of Intel Corporation.

SPARCstation is trademark of SPARC International, Inc., but licensed exclusively to Sun Microsystems, Inc.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X Window System is a trademark of the Massachusetts Institute of Technology.

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/mirrors/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 Software Tools With SunOS

This chapter helps you install release 2.00 of the TMS320C8x software tools on a SPARCstation™ running SunOS™ version 4.1.x (or higher). The SunOS version of the TMS320C8x software tools package is composed of the following items:

- Master processor (MP) and parallel processor (PP) C compilers
- MP and PP assemblers
- Linker
- MP and PP simulators
- MP and PP C source debuggers
- Runtime-support and C I/O libraries

Topic **Page**

1.1 System Requirements

To install and use the ’C8x software tools, you need the items listed in the following hardware and software checklists.

Hardware checklist

☐	**Host**	SPARCstation or compatible system with SPARCstation 2 class or higher performance
☐	**Memory**	32M bytes of RAM
☐	**Disk space**	16M bytes of disk space for the software tools. In addition, you may need 1G bytes or more for software development or for working with digital images.
☐	**Display**	Color monitor
☐	**Required hardware**	Mouse
☐		Keyboard
☐		CD-ROM drive

Software checklist

☐	**Operating system**	SunOS version 4.1.3 (or higher) or SunOS version 5.x (also known as Solaris™ 2.x) using an X Window System™ window manager, such as OpenWindows™ version 3.0 (or higher). If you use SunOS 5.x, you must have the Binary Compatibility Package (BCP) installed; if you do not have this package, get your system administrator’s help.
☐	**Root privileges**	If you are running SunOS 4.1.x, 5.0, or 5.1, you *must* have root privileges to mount and unmount the CD-ROM. If you do not, get help from your system administrator.
☐	**Interprocess communication features**	Interprocess communication (IPC) features are included with your operating system. To verify that you have the IPC features enabled, enter **ipcs** from the command line; if you have the IPC features installed, you will see a series of messages about shared memory and semaphores.
☐	**Make utility**	If you have SunOS 5.x, you must install the UNIX™ make utility.
☐	**CD-ROM**	*TMS320C8x (MVP) Online Reference*
☐	**CD-ROM**	*TMS320C8x Software Toolkit*

1.2 Before You Use the TMS320C8x Simulator and Debuggers

You install the parallel debug manager (PDM), the MP debugger, the PP debugger, and the simulator core as separate files and execute them as individual tasks; however, these tasks work together to form the ’C8x simulation environment. This environment uses the interprocess communication (IPC) features of UNIX (shared memory, message queues, and semaphores) to manage communications between the different tasks that make up the simulator. If you are not sure whether the IPC features are enabled, see your system administrator.

To use the ’C8x simulation environment, you should be familiar with the IPC status (ipcs) and IPC remove (ipcrm) UNIX commands. If you use the UNIX kill (send signal) command to terminate execution of the simulator tasks, you also need to use the ipcrm command to remove the shared memory, message queues, and semaphores used by the simulator.

1.3 Installing the Tools

This section explains the process of installing the software tools on your hard-disk system. The software package is shipped on a CD-ROM. To install the software, you must mount the CD-ROM, copy the files, and unmount the CD-ROM.

Note: Root Privileges

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

Mounting the CD-ROM

The steps to mount the CD-ROM vary according to your operating-system version:

- ❑ If you have 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 ⏎
  ```

- ❑ 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 ⏎
  ```

- ❑ If you have SunOS 5.2 or higher:

  - If your CD-ROM drive is already attached, load the CD-ROM into the drive and enter the following from a command shell:

    ```
    cd /cdrom/cdrom0 ⏎
    ```

  - If you do not have a CD-ROM drive attached, you must shut down your system to the PROM level, attach the CD-ROM drive, and enter the following:

    ```
    boot –r ⏎
    ```

    After you log into your system, load the CD-ROM into the drive and enter the following from a command shell:

    ```
    cd /cdrom/cdrom0 ⏎
    ```

Copying the files

After you mount the CD-ROM, you must create the directory that will contain the software tools and copy the tools to that directory.

1) Create a directory on your hard disk for the software tools. To create this directory, enter:

   mkdir *pathname*/**c8xtools** ⏎

2) Copy the files from the CD-ROM to your hard disk system:

   cp –r * *pathname*/**c8xtools** ⏎

Unmounting the CD-ROM

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

- ❑ If you have SunOS 4.1.x, SunOS 5.0, or SunOS 5.1, as root, enter the following from a command shell:

  cd ⏎
  umount /cdrom ⏎
  eject /dev/sr0 ⏎
  exit ⏎

- ❑ If you have SunOS 5.2 or higher, enter the following from a command shell:

  cd ⏎
  eject ⏎

1.4 Setting Up the Environment

You can define *environment variables* to set certain software tool parameters that 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 remainder of this section describes the environment variables that the software tools use. You can set up the environment variables in your .login or .cshrc file (for C shells) or .profile file (for Bourne or Korn shells).

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

You must include the software tools bin directory in your path statement. This allows you to specify the software 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:

  pathname`/c8xtools/bin`

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

  - For C shells:

    `set path=($path` *`pathname`*`/c8xtools/bin)`**

  - For Bourne or Korn shells:

    `PATH=$PATH` *`pathname`*`/c8xtools/bin`**

    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. Set up the A_DIR variable to identify the `c8xtools/lib` and `c8xtools/exec/lib` directories:

- For C shells:

```
setenv A_DIR "pathname/c8xtools/lib;pathname/c8xtools/exec/lib"
```

- For Bourne or Korn shells:

```
A_DIR="pathname/c8xtools/lib;pathname/c8xtools/exec/lib"
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 *TMS320C80 (MVP) Code Generation Tools 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. Set up the C_DIR variable to identify the `c8xtools/lib` and `c8xtools/exec/lib` directories:

- For C shells:

```
setenv C_DIR "pathname/c8xtools/lib;pathname/c8xtools/exec/lib"
```

- For Bourne or Korn shells:

```
C_DIR="pathname/c8xtools/lib;pathname/c8xtools/exec/lib"
export C_DIR
```

You can separate pathnames with a semicolon or with blanks. Once you set C_DIR, you can use the #include directive in your C source code without specifying path information.

Identifying alternate directories for the debugger (D_DIR)

By default, the debugger searches the current directory for auxiliary files such as init.cmd, init.pdm, and init.clr that are needed to use the debugger and PDM. Use the D_DIR environment variable to define additional search paths. Set up the D_DIR variable to identify the `c8xtools/lib` directory:

- For C shells:

  setenv D_DIR "*pathname***/c8xtools/lib"**

- For Bourne or Korn shells:

  D_DIR="*pathname***/c8xtools/lib"**
  export D_DIR

Identifying directories that contain program source files (D_SRC)

By default, the debugger searches the current directory for program source files that are accessed from the debugger. Use the D_SRC environment variable to define additional search paths. The general format for the D_SRC variable is:

- For C shells:

  setenv D_SRC "*pathname1;* [*pathname2...*]**"**

- For Bourne or Korn shells:

  D_SRC="*pathname1;* [*pathname2...*]**"**
  export D_SRC

For example, if your 'C8x program source files were located in the directory `/user/fred/c8xcode`, the D_SRC setup would be:

```
setenv D_SRC "/user/fred/c8xcode"
```

Setting default shell options (C_OPTION)

When using the program shell (mpcl or ppcl), 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 the input filenames, the program shell reads the contents of the C_OPTION environment variable. The options and/or input filenames that you define with C_OPTION are used every time you run the program shell.

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 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 *TMS320C80 (MVP) Code Generation 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              ;
```

Setting default debugger options (D_OPTIONS)

You might find it useful to set default options for the debugger using the D_OPTIONS environment variable. When you use the D_OPTIONS environment variable, the debugger uses the default options and/or input filenames that you define with D_OPTIONS every time you invoke the debugger.

The syntax for the D_OPTIONS environment variable is:

- For C shells:

  setenv D_OPTIONS "[*object filename*] [*debugger options*]**"**

- For Bourne or Korn shells:

  D_OPTIONS="[*object filename*] [*debugger options*]**"**
  export D_OPTIONS

Each time you invoke the debugger, the specified object file will be loaded and the specified options will be used. Table 1–1 lists the options that you can identify with D_OPTIONS.

Table 1–1. Options for Use With D_OPTIONS

Option	Description
–b[b]	Select the screen size
–d *machinename*	Display debugger on different machine (X Windows only)
–i *pathname*	Identify additional directories
–min	Select the minimal debugging mode
–mv *version*	Select the device version
–n *processorname*	Identify the name of the processor
–o	Enable C I/O
–s	Load the symbol table only
–t *filename*	Identify a new initialization file
–v	Load without the symbol table

Note that you can override D_OPTIONS by invoking the debugger with the –x option.

For more information about options, see the *TMS320C80 (MVP) C Source Debugger User's Guide*.

Displaying the debugger on a different machine (LD_LIBRARY_PATH, DISPLAY)

If you are using the X Window System, you can display the debugger on a different machine than the one the parallel debug manager and simulator core are running on. To do so, you need to set up two environment variables:

- ❑ Be sure that the LD_LIBRARY_PATH environment variable is set to the following:

  ```
  LD_LIBRARY_PATH $OPENWINHOME/lib
  ```

  If the LD_LIBRARY_PATH variable is not set correctly, use one of these commands:

  - For C shells:

    ```
    setenv LD_LIBRARY_PATH "$OPENWINHOME/lib"
    ```

  - For Bourne or Korn shells:

    ```
    LD_LIBRARY_PATH="$OPENWINHOME/lib"
    export LD_LIBRARY_PATH
    ```

- ❑ Set up the DISPLAY environment variable. The general format for doing this is:

  - For C shells:

    setenv DISPLAY "*machinename***:0"**

  - For Bourne or Korn shells:

    DISPLAY="*machinename***:0"**
    export DISPLAY

Note: IP address

You may need to specify the IP address of your machine in place of its machinename if the debugger does not display properly.

You can also specify a different machine by using the –d debugger option (see the *TMS320C80 (MVP) C Source Debugger User's Guide* for more information). If you use both the DISPLAY environment variable and –d, the –d option overrides DISPLAY.

1.5 Using the Debugger With the X Window System

When you're using the X Window System to run the 'C8x debugger, you need to know about the keyboard's special keys, the debugger fonts, and using the debugger on a monochrome monitor.

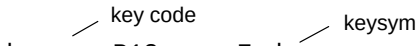
Using the keyboard's special keys

The debugger uses some special keys that you can map differently from your particular keyboard. Some keyboards, such as the Sun Type 5 keyboard, have these special symbols on separate keys. Other keyboards, such as the Sun Type 4 keyboard, do not have the special keys, but the functions are available.

The special keys that the debugger uses are shown in the following table with their corresponding keysym. A *keysym* is a label that interprets a keystroke; it allows you to modify the action of a key on the keyboard.

Key	Keysym
F1 to F10	F1 to F10
PAGE UP	Prior
PAGE DOWN	Next
HOME	Home
END	End
INSERT	Insert
→	Right
←	Left
↑	Up
↓	Down

Use the X utility xev to check the keysyms associated with your keyboard. If you need to change the keysym definitions, use the xmodmap utility. For example, you could create a file that contains the following commands and use that file with xmodmap to map a Sun Type 4 keyboard to the keys listed above:




```
keysym R13    = End
keysym Down   = Down
keysym F35    = Next
keysym Left   = Left
keysym Right  = Right
keysym F27    = Home
keysym Up     = Up
keysym F29    = Prior
keysym Insert = Insert
```

Refer to your X Window System documentation for more information about using xev and xmodmap.

Changing the debugger font

You can change the font of the debugger screen by using the xrdb utility and modifying the .Xdefaults file in your root directory. For example, to change the fonts of the MP and PP debuggers to Courier, add the following line to the .Xdefaults file:

```
mpsim*font:courier
ppsim*font:courier
```

For more information about using xrdb to change the font, refer to your X Window System documentation.

Color mappings on monochrome screens

Although a color monitor is recommended (and necessary for the graphic display features), the following table shows the color mappings for monochrome screens:

Color	Appearance on Monochrome Screen
black	black
blue	black
green	white
cyan	white
red	black
magenta	black
yellow	white
white	white

1.6 Where to Go From Here

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

- ❑ Go to Chapter 3, *Verifying the Installation: A Walkthrough of the Software Tools*. This chapter helps you verify your installation and provides you with an overview of how to invoke and use the assembler, linker, and compiler.
- ❑ Read Chapter 4, *Release Notes*. This chapter explains the new features included in release 2.00 of the software tools.
- ❑ Use the *TMS320C8x (MVP) Online Reference* for more information about how to use the ’C8x tools.

Chapter 2

Setting Up the Software Tools With Windows NT or Windows 95

This chapter helps you install release 2.00 of the TMS320C8x software tools on a 32-bit x86-based or Pentium™ PC™ running Windows NT™ or Windows 95. The Windows NT and Windows 95 version of the TMS320C8x software tools package is composed of the following items:

- Master processor (MP) and parallel processor (PP) C compilers
- MP and PP assemblers
- Linker
- Runtime-support and C I/O libraries

Notice that the simulators and debuggers are not included in the Windows NT and Windows 95 version of the TMS320C8x software tools.

Topic **Page**

2.1 System Requirements

To install and use the ’C8x software tools, you need the items listed in the following hardware and software checklists.

Hardware checklist

☐	**Host**	32-bit x86-based or Pentium-based PC
☐	**Memory**	Minimum of 16M bytes of RAM plus 12M bytes of available hard-disk space
☐	**Display**	Monochrome or color monitor (color recommended)
☐	**Required hardware**	CD-ROM drive
☐	**Optional hardware**	Microsoft™-compatible mouse

Software checklist

☐	**Operating system**	Windows NT Workstation version 3.5 or later Windows 95 version 4.0 or later
☐	**CD-ROM**	*TMS320C8x (MVP) Online Reference*
☐	**CD-ROM**	*TMS320C8x Software Toolkit*

2.2 Installing the Tools

The software 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 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.

If you choose not to have the environment variables set up automatically, see Section 2.3, *Setting Up the Environment,* for alternate ways that you can set up the environment variables.

2.3 Setting Up the Environment

You can define *environment variables* to set certain software tool parameters that 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.

By default, the installation program sets up the following environment variables:

```
SET PATH=C:\C8XTOOLS;%PATH%
SET A_DIR=C:\C8XTOOLS\LIB; C:\C8XTOOLS\EXEC\LIB
SET C_DIR=C:\C8XTOOLS\LIB; C:\C8XTOOLS\EXEC\LIB
```

The remainder of this section describes these and other environment variables, and discusses alternative ways that they 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)

Under Windows NT or Windows 95, you must include the `c8xtools` directory in your PATH statement. This allows you to specify the software 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 beginning of the PATH statement:

  ```
  C:\C8XTOOLS\BIN;
  ```

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

  ```
  SET PATH=C:\C8XTOOLS\BIN;%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. Under Windows NT or Windows 95, set up the A_DIR variable to identify the `c8xtools\lib` and `c8xtools\exec\lib` directories:

```
SET A_DIR=C:\C8XTOOLS\LIB;C:\C8XTOOLS\EXEC\LIB
```

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 *TMS320C80 (MVP) Code Generation Tools 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. Under Windows NT or Windows 95, set up the C_DIR variable to identify the `c8xtools\lib` and `c8xtools\exec\lib` directories:

```
SET C_DIR=C:\C8XTOOLS\LIB;C:\C8XTOOLS\EXEC\LIB
```

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 program shell (mpcl or ppcl), 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 the input filenames, the program shell reads the contents of the C_OPTION environment variable. The options and/or input filenames that you define with C_OPTION are used every time you run the program shell.

Under Windows NT or Windows 95, 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 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 *TMS320C80 (MVP) Code Generation 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
```

2.4 Where to Go From Here

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

- ❑ Go to Chapter 3, *Verifying the Installation: A Walkthrough of the Software Tools*. This chapter helps you verify your installation and provides you with an overview of how to invoke and use the assembler, linker, and compiler.
- ❑ Read Chapter 4, *Release Notes*. This chapter explains the new features included in release 2.00 of the software tools.
- ❑ Use the *TMS320C8x (MVP) Online Reference* for more information about how to use the ’C8x tools.

Chapter 3

Verifying the Installation: A Walkthrough of the Software Tools

This chapter provides a quick walkthrough of the ’C8x software tools so that you can verify your installation and start compiling, assembling, and linking code immediately. For more information about using the code generation tools, refer to the *TMS320C80 (MVP) Code Generation Tools User’s Guide.*

Topic **Page**

3.1 Assembler and Linker Walkthrough

Two tools that you will probably use often are the assembler and linker. This section helps you get started with these tools.

Step 1: Create two source files

Create the source files shown in Example 3–1 and Example 3–2; name them mp_a.asm and pp_a.s, respectively.

Example 3–1. mp_a.asm File

```
              .global   PP0_START
PP0_PARM      .set      0x010001b8
PP0_UNHALT    .set      0x30004001

MP_start:     add       –1,r0,r1            ; clears INTPEN
              wrcr      INTPEN, r1

              or        PP0_START,r0,r1
              st        PP0_PARM(r0),r1     ; PP0 start addr
              or        PP0_UNHALT, r0, r1  ; unhalt PP0
              cmnd      r1

              add       r0,r0,r1            ; clear r1
loop:         br        loop
              add       1, r1, r1           ; inc r1
```

Example 3–2. pp_a.s File

```
              .global   PP0_START

PP_DATA       .set      0x01020304
INCREMENT     .set      0x01010101
PP_ADDR       .set      0x01003400

PP0_START:    a9 = PP_ADDR
              d0 = PP_DATA
              d1 = INCREMENT
              lrse0 = 15
              nop
              nop
Loop1:        d0 = d1 + d0        ; inc data
               || *a9++ =ub d0    ; store data
              br = PP0_START
              nop
              nop
```

Step 2: Assemble the mp_a.asm file

To assemble the mp_a.asm file, enter:

`mpasm mp_a` ↵

The mpasm command invokes the MP assembler. You don't have to specify the .asm file extension for the input source file (mp_a.asm), because the MP assembler uses .asm as the default extension. This example creates an object file called mp_a.obj. The assembler creates an object file if no errors are encountered during assembly.

Step 3: Assemble the pp_a.s file

To assemble the pp_a.s file, enter:

ppasm pp_a –l ↵

The ppasm command invokes the PP assembler. You don't have to specify the .s file extension for the input source file (pp_a.s), because the PP assembler uses .s as the default extension. This example creates an object file called pp_a.o.

The –l (lowercase L) option tells the PP assembler to create a listing file. The listing for this example is called pp_a.lst (shown in Example 3–3). Creating a listing file is not required; however, a listing file provides you with more information and allows you to verify whether or not the assembly has resulted in the object code that you intended.

Example 3–3. pp_a.lst File

```
TMS320C8x PP Macro Assembler     Version 2.00     Tue Oct 15 14:30:20 1996
Copyright (c) 1993–1996    Texas Instruments Incorporated
pp_a.s                                                          PAGE    1

       1                                        .global PP0_START
       2
       3          0000000001020304  PP_DATA      .set    0x01020304
       4          0000000001010101  INCREMENT    .set    0x01010101
       5          0000000001003400  PP_ADDR      .set    0x01003400
       6
       7 00000000 9B8118C001003400  PP0_START:   a9 = PP_ADDR
       8 00000008 9B801A4001020304               d0 = PP_DATA
       9 00000010 9B811A4001010101               d1 = INCREMENT
      10 00000018 9B80078000344100               lrse0 = 15
      11 00000020 8800000000104100               nop
      12 00000028 8800000000104100               nop
      13 00000030 9A8010800050C008  Loop1:       d0 = d1 + d0     ; inc data
      14                                        || *a9++ =ub d0  ; store data
      15 00000038 97811BC000000000'             br = PP0_START
      16 00000040 8800000000104100               nop
      17 00000048 8800000000104100               nop

 No Errors, No Warnings
```

Step 4: Link the mp_a.obj and pp_a.o files

To link the mp_a.obj and pp_a.o files, enter:

mvplnk mp_a.obj pp_a.o –m link_a.map –o prog_a.out ↵

The mvplnk command invokes the 'C8x linker. The linker combines the input object files, mp_a.obj and pp_a.o, to create an executable object module called prog_a.out. The –o linker option specifies the linked module name. The –m option creates a map output file from the linking operation. Example 3–4 shows the map file resulting from this example.

Example 3–4. link_a.map File

```
*****************************************************
TMS320C8x COFF Linker              Version 2.00
*****************************************************
Wed Oct 16 13:43:48 1996

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


MEMORY CONFIGURATION

   name     origin    length      used    attributes   fill
 --------  --------  ---------  --------  ----------  ------
 DRAMS     00000004  000000ffc  00000000     RWIX
 DRAM2     00008000  000000800  00000000     RWIX
 PRAM      01000200  000000600  00000000     RWIX
 EXTMEM    02000000  000080000  00000080     RWIX

SECTION ALLOCATION MAP

 output                                  attributes/
section   page    origin      length       input sections
--------  ----  ----------  ----------   ----------------
.text      0    02000000    00000030
                  02000000    00000030     mp_a.obj (.text)
                  02000030    00000000     pp_a.o (.text)

.ptext     0    02000030    00000050
                  02000030    00000050     pp_a.o (.ptext)

.bss       0    02000000    00000000     UNINITIALIZED
                  02000000    00000000     mp_a.obj (.bss)
                  02000000    00000000     pp_a.o (.bss)

.data      0    02000000    00000000     UNINITIALIZED
                  02000000    00000000     mp_a.obj (.data)
                  02000000    00000000     pp_a.o (.data)

.pbss      0    00000004    00000000     PASS SECTION
```

Example 3–4.link_a.map File (Continued)

```
GLOBAL SYMBOLS

address  name                                address  name
-------- ----                                -------- ----
02000000 .bss                                02000000 end
02000000 .data                               02000000 .data
02000000 .text                               02000000 edata
02000030 PP0_START                           02000000 .bss
02000000 edata                               02000000 .text
02000000 end                                 02000030 PP0_START
02000030 etext                               02000030 etext

[7 symbols]
```

3.2 C Compiler Walkthrough

The TMS320C8x C compilers operate in three passes.

- The first pass parses the code.
- The second pass optionally optimizes the code.
- The third pass produces a single assembly language source file that must be assembled and linked.

Use the shell program, which is included with the compilers, to compile, assemble, and link a C program. This section helps you get started with compiling C programs.

Step 1: Create a sample C file

Create a sample file called function.c that contains the code shown in Example 3–5.

Example 3–5. function.c File

```
/************************************/
/*           function.c             */
/*  (sample file for walkthrough)    */
/************************************/
main ()
{
   int x = –3;
   x = abs_func(x);
}

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

Step 2: Compile and assemble function.c

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

```
mpcl function.c ↵
```

As the shell program compiles, it displays the information shown in Example 3–6.

Example 3–6. Shell Program Messages

```
[function.c]
TMS320C8x MP ANSI C Compiler     Version 2.00
Copyright (c) 1993–1996    Texas Instruments Incorporated
   ″function.c″   ==> main
   ″function.c″   ==> abs_func
TMS320C8x MP ANSI C Codegen      Version 2.00
Copyright (c) 1993–1996    Texas Instruments Incorporated
   ″function.c″:  ==> main
   ″function.c″:  ==> abs_func
TMS320C8x MP Macro Assembler     Version 2.00
Copyright (c) 1993–1996    Texas Instruments Incorporated
 PASS 1
 PASS 2

 No Errors,  No Warnings
```

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

- ❏ mpac → MP C parser
- ❏ mpcg → MP code generator
- ❏ mpasm → MP assembler

Inspecting the assembly language output

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

```
mpcl function.c –k ↵
```

Changing the output file

Also by default, the shell program creates a COFF object file as output; however, if you use the –z option, the shell program will invoke the linker to produce an executable COFF object module. The following examples show the two ways of creating an executable object module:

- Using the linker directly, link the object file with the runtime-support library mp_rts.lib:

  mvplnk –c function –o function.out –l mp_rts.lib ↵

  This example uses the –c linker option because the code came from a C program. The –l option tells the linker that the input file mp_rts.lib is an object library. The –o option names the output module function.out. If you don't use the –o option, the linker uses a.out as the default name.

- Using the shell program, the –z option tells the shell program to run the linker. All other linker options must follow –z.

  mpcl function.c –z –o function.out –l mp_rts.lib ↵

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

  - mpac → MP C parser
  - mpcg → MP code generator
  - mpasm → MP assembler
  - mvplnk → 'C8x linker

Using the interlist utility

The TMS320C8x 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 –s option:

mpcl function.c –s –z –o function.out –l mp_rts.lib ↵

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

Chapter 4

Release Notes

This chapter contains documentation of tools and features that are new or have been changed since the last release. It is not an exhaustive list of all code changes since the last release, but it is a list of all of the changes that may require modifications to your C source, assembly source, linker control files, debugger batch or initialization files, or makefiles. A list of defects that are known to exist in this release of the tools is also included. Whenever possible, work-arounds are provided.

Topic **Page**

4.1 Media Contents

The TMS320C8x software tools are supported on SPARCstations with SunOS and on PCs with Windows NT or Windows 95.

Table 4–1 lists the top-level directories that are installed and briefly describes the contents of each directory.

Table 4–1. Contents of Top-Level Installed Directory

Directory	Description
app_code	Sample applications and demonstration code
bin	Executable programs
exec	Multitasking executive libraries, documentation, and source code (For more information, see the file readme.txt in this directory.)
lib	Command files, header files, and object libraries for C I/O and runtime support
src	Source code for C I/O and run-time support

The file fixbugs.txt is also located in the top-level directory. The contents of this file describe the problems that have been fixed since the last release.

Table 4–2 through Table 4–6 list the files shipped on the *TMS320C8x Software Toolkit* CD-ROM.

Table 4–2. Contents of app_code Directory

Directory	Description
3x3filtr	3x3 FIR Graphical Filter Demo
cif2rgb	Converts images in YCbCr 411 format to XBGR or XRGB format
cio	Demonstrations of building and debugging programs with C I/O on the MP and PP
demo	Graphical example using the multitasking executive and transfer controller
dsplib	Library of C-callable highly optimized DSP routines
dtmf	Modified Goertzel algorithm in DTMF (touchtone) detection
execdemo	Extensive graphical example using the multitasking executive and transfer controller
execsmpl	Simple example using the multitasking executive to run a program on the PP
ffts	64-point FFT for the MP and a 256-point FFT for the PP
grafix16	A set of 16-bit PP graphics routines and associated demonstration programs
image	A directory of image-related demonstration programs. The image directory contains the following subdirectories: median — PP Median filtering graphical demonstration roberts — Roberts edge-detection simulation on the PP thresh — Example of a thresholding tight loop performed on the PP
xform	3x3 and 4x4 floating-point matrix multiply operations for the MP

Table 4–3. Contents of bin Directory

Windows NT/95 File	SunOS File	Description
clist.exe	clist	C source interlist utility
mpac.exe	mpac	MP ANSI C parser
mpasm.exe	mpasm	MP assembler
mpcg.exe	mpcg	MP ANSI C code generator
mpcl.exe	mpcl	MP compiler shell program
mpmk.exe	mpmk	MP library build utility
mpopt.exe	mpopt	MP optimizer
mvpar.exe	mvpar	’C8x archiver
mvphex.exe	mvphex	’C8x hex conversion utility
mvplnk.exe	mvplnk	’C8x COFF linker
ppac.exe	ppac	PP ANSI C parser
ppasm.exe	ppasm	PP assembler
ppca.exe	ppca	PP code compactor
ppcg.exe	ppcg	PP code generator
ppcl.exe	ppcl	PP compiler shell program
ppmk.exe	ppmk	PP library build utility
ppopt.exe	ppopt	PP optimizer
Simulator and Debugger Files (SPARCstations only):		
N/A	pdm	Parallel debug manager environment
N/A	mpsim	MP C source debugger and simulator core
N/A	ppsim	PP C source debugger and simulator core
N/A	simMVP	’C8x simulator core (the mpsim and ppsim files invoke simMVP)

Table 4–4. Contents of exec Directory

Directory	Description
doc	Detailed documentation on how to use the multitasking executive
lib	Header and library files for the multitasking executive
src	Source code for the multitasking executive libraries

Table 4–5. Contents of lib Directory

File	Description		
mp_cio.lib	MP ANSI C standard I/O library—big endian		
mp_ciol.lib	MP ANSI C standard I/O library—little endian		
pp_cio.lib	PP ANSI C standard I/O library—big endian		
pp_ciol.lib	PP ANSI C standard I/O library—little endian		
mp_rts.lib	MP runtime-support functions—big endian		
mp_rtsl.lib	MP runtime-support functions—little endian		
pp_rts.lib	PP runtime-support functions—big endian		
pp_rtsl.lib	PP runtime-support functions—little endian		
#include header files for RTS:			
*.h, *.i	assert.h	ctype.h	errno.h
	file.h	float.h	format.h
	limits.h	math.h	mvp.h
	setjmp.h	stdarg.h	stddef.h
	stdio.h	stdlib.h	string.h
	time.h	trgcio.h	packetpp.i
Sample linker conrol files:			
mpcio.cmd	MP ANSI C standard I/O command file—big endian		
mpciol.cmd	MP ANSI C standard I/O command file—little endian		
ppcio.cmd	PP ANSI C standard I/O command file—big endian		
ppciol.cmd	PP ANSI C standard I/O command file—little endian		
mplnk.cmd	MP runtime-support functions—big endian		
mplnkl.cmd	MP runtime-support functions—little endian		
pplnk.cmd	PP runtime-support functions—big endian		
pplnkl.cmd	PP runtime-support functions—little endian		

Table 4–5. Contents of lib Directory (Continued)

File	Description
Simulator and Debugger Files (SPARCstations only):	
init.cmd	A general-purpose batch file that contains debugger commands. This batch file, shipped with the debugger, defines an MP and PP memory map. If this file is not found when you invoke the debugger, then all memory is invalid at first. When you first start using the debugger, this memory map should be sufficient for your needs. Later, you may want to define your own memory map. For information about setting up your own memory map, refer to the *Defining a Memory Map* chapter in the *TMS320C80 (MVP) C Source Debugger User's Guide*.
initc80.cmd	Batch file to configure the 'C80 memory map. This file is identical to init.cmd.
initc82.cmd	Batch file to configure the 'C82 memory map. This file can be used in place of the default init.cmd by specifying the option –t initc82.cmd when invoking the debugger.
init.pdm	A general-purpose batch file that contains special parallel debug manager commands. These commands allow you to group and send commands to debuggers under the control of the parallel debug manager. The parallel debug manager reads this file during invocation by default.
initc80.pdm	Batch file to spawn the 'C80 debuggers. This file is identical to init.pdm.
initc82.pdm	Batch file to spawn the 'C82 debuggers. This file can be used in place of the default init.pdm by specifying –t initc82.pdm when invoking PDM.
init.clr	A general-purpose screen configuration file. If init.clr is not found when you invoke the debugger, the debugger uses the default screen configuration. For information about this file and about setting up your own screen configuration, refer to the *Customizing the Debugger Display* chapter in the *TMS320C80 (MVP) C Source Debugger User's Guide.*
siminit.cmd	A general-purpose screen configuration file. The simulator will read this file instead of init.cmd when siminit.cmd exists in the D_DIR path. If init.clr is not found when you invoke the debugger, the debugger uses the default screen configuration. For information about this file and about setting up your own screen configuration, refer to the *Customizing the Debugger Display* chapter in the *TMS320C80 (MVP) C Source Debugger User's Guide.*

Table 4–6. Contents of src Directory

File	Description
mp_rts.src	Source library for mp_rts.lib and mp_rtsl.lib
pp_rts.src	Source library for pp_rts.lib and pp_rtsl.lib
mp_cio.src	Source library for mp_cio.lib and mp_ciol.lib
pp_cio.src	Source library for pp_cio.lib and pp_ciol.lib

4.2 Changes to the MP and PP Simulators (SPARCstations Only)

The following subsections detail enhancements made to release 2.00 of the MP and PP simulators.

Selecting little-endian support (–me option)

The MP and PP simulators include little-endian support. The –me debugger option causes the simulator to simulate the ’C8x as a little-endian target. To simulate a little-endian target, you must specify the –me option when you invoke the debugger:

- For the MP debugger:

  mpsim –me

- For the PP debugger:

  ppsim –me –n *processorname*

Selecting the minimal debugging mode (–min option)

The debugger automatically displays whatever code is currently running: assembly language or C. Depending on the code that is currently running, the debugger displays various windows, such as the DISASSEMBLY, COMMAND, CPU, MEMORY, or CALLS window.

The debugger has a *minimal* debugging mode that displays the COMMAND, WATCH, and DISP windows only. The WATCH and DISP windows are displayed only if you cause them to display (by entering the WA or DISP commands). Minimal mode may be useful when you need to debug a memory problem.

To invoke the debugger and enter minimal mode, use the –min option:

mpsim –min

Invoking the ’C82 version of the debugger (–mv option)

The MP and PP simulators now support the ’C82. To simulate the ’C82, you must specify the –mv82 option when you invoke the debugger:

- For the MP debugger:

  mpsim –mv82

- For the PP debugger:

  ppsim –mv82 –n *processorname*

Note: The TMS320C82 Has Only Two Parallel Processors

Keep in mind that the ’C82 has only two PPs. If you use the –mv82 option and specify a processor name other than mvp1_pp0 or mvp1_pp1, the debugger displays an error. You cannot access mvp1_pp2 or mvp1_pp3 when you use the –mv82 option.

You can also use the –mv82 option when you invoke the debuggers from the parallel debug manager (PDM):

spawn mpsim –mv82

In a single PDM session, when you spawn one processor simulator with the –mv82 option, the PDM automatically assumes that all future processors will also be ’C82. For example, you could enter the following from the PDM prompt:

```
PDM> spawn mpsim –mv82
PDM> spawn ppsim –n mvp1_pp0
PDM> spawn ppsim –n mvp1_pp1
```

This command sequence invokes the ’C82 version of the MP debugger and the ’C82 version of two PP debuggers.

When invoking the debugger from PDM, you must use the correct initialization file for your target processor. The next section describes how to set up a target-specific initialization file.

New initialization files for the parallel debug manager

When you invoke the parallel debug manager (PDM), it searches for a file called init.pdm. The PDM looks first in the current directory and then in the directories defined in the D_DIR environment variable . When the PDM finds this file, it reads and executes the commands in the file.

The default init.pdm file shipped with version 2.00 contains initialization commands that set up the PDM for use with the ’C80. If you want to use the PDM with the ’C82, you can specify the –t option when you invoke the PDM:

pdm –t initc82.pdm

The initc82.pdm contains initialization commands that set up the PDM for use with the ’C82.

If you plan to use the PDM with the ’C82 on a regular basis, you can rename the initc82.pdm file to init.pdm. PDM will then default to ’C82 instead of ’C80. To switch to the ’C80 initialization file, use the –t option and specify the initc80.pdm file:

pdm –t initc80.pdm

Entering commands from the command line

The editor for the debugger command line has been enhanced. Once you have typed a command, you can edit the text on the command line with these keystrokes:

To...	Press...
Move back over text without erasing characters	← †
Move forward through text without erasing characters	→ † or CONTROL L
Move to the beginning of the previous word without erasing characters	CONTROL ← †
Move to the beginning of the next word without erasing characters	CONTROL → †
Move to the beginning of the line without erasing characters	ALT ← †
Move to the end of the line without erasing characters	ALT → †
Move back over text while erasing characters	CONTROL H or BACK SPACE or DEL
Move forward through text while erasing characters	SPACE
Insert text into the characters that are already on the command line	INSERT
Delete text from the right of the cursor position	CONTROL K

† You can use the arrow keys only when the COMMAND window is selected.

Note:

1) When the COMMAND window is not active, you cannot use the arrow keys to move through or edit text on the command line.
2) Typing a command doesn't make the COMMAND window the active window.

Using multiple MEMORY windows

You can now open more than four MEMORY windows. The MEM command has a new optional *window name* parameter. When you open an additional MEMORY window, the debugger appends the *window name* to the MEMORY window label. You can create as many MEMORY windows as you need. The basic syntax for the MEM command is:

mem *expression* [*, display format*] [, *window name*]

You can use the MEM command to display a different memory range in a window. The *window name* parameter is optional if you are displaying a different memory range in the default MEMORY window. Use the *window name* parameter when you want to display a new memory range in one of the additional MEMORY windows.

Using multiple WATCH windows

You can now access multiple WATCH windows. Use the *window name* parameter as described for each WATCH window command.

- The WA command has a new optional *window name* parameter. When you open an additional WATCH window, the debugger appends the *window name* to the WATCH window label. You can create as many WATCH windows as you need. The basic syntax for the WA command is:

  wa *expression* [,[*label*] [*,* [*display format*] [, *window name*]]]

  If you omit the *window name* parameter, the debugger displays the expression in the default WATCH window (labeled WATCH).

- The WD command deletes a specific item from the WATCH window. The WD command's *index number* parameter must correspond to one of the watch indexes listed in the WATCH window. The optional *window name* parameter is used to specify a particular WATCH window. The basic syntax for the WD command is:

  wd *index number* [, *window name*]

- The WR command deletes all items from a WATCH window and closes the window.

  - To close the default WATCH window, enter:

    wr

  - To close one of the additional WATCH windows, use this syntax:

    wr *windowname*

  - To close all WATCH windows, enter:

    wr *

New debugger commands

The debugger supports the following new commands on all TI processor platforms.

cd, chdir — *Change Directory*

Syntax

cd [*directory name*]
chdir [*directory name*]

Menu selection

none

Environments

☑ basic debugger ☐ PDM ☐ profiling

Description

The CD or CHDIR command changes the current working directory from within the debugger. You can use relative pathnames as part of the *directory name*. If you don't use a *directory name*, the CD command displays the name of the current directory. Note that this command can affect any other command whose parameter is a filename, such as the FILE, LOAD, and TAKE commands, when used with the USE command. For example:

```
cd /usr/local/bin
cd ../../bin
cd /home
```

dir — *List Directory Contents*

Syntax

dir [*directory name*]

Menu selection

none

Environments

☑ basic debugger ☐ PDM ☐ profiling

Description

The DIR command displays a directory listing in the display area of the COMMAND window. If you use the optional *directory name* parameter, the debugger displays a list of the specified directory's contents. If you don't use a *directory name*, the debugger lists the contents of the current directory.

You can list only files that match a specific format within a directory by using the asterisk (*) wildcard character. If the *directory name* ends in a partial filename with an asterisk, the debugger lists only the files which match the wildcard string. For example, to list every file in the home directory that has a .cmd extension, you would enter:

```
DIR /home/*.cmd
```

safehalt — *Toggle Safehalt Mode*

Syntax **safehalt** {**on** | **off**}

Menu selection none

Environments ☑ basic debugger ☑ PDM ☐ profiling

Description The SAFEHALT command places the debugger in safehalt mode. When safehalt mode is off (the default), you can halt a running simulator either by pressing ESC or by clicking a mouse button. When safehalt mode is on, you can halt a running simulator only by pressing ESC ; mouse clicks are ignored.

Simulating external interrupts (MP simulator only)

The MP simulator allows you to simulate and monitor four interrupt signals and to specify at what clock cycle you want an interrupt to occur. To do this, you create a data file and connect it to one of the interrupt levels (or *pins*): EINT1, EINT2, EINT3, or LINT4.

> **Note:**
>
> The time interval is expressed as a function of CPU clock cycles. Simulation begins at the first clock cycle.

In order to simulate interrupts, you must first set up an input file that lists interrupt intervals. Your file must contain a clock cycle in the following format:

[(] [+] *cycle* [)] [**rpt** {*n* | **EOS**}] [[(] [+] *cycle* [)] [**rpt** {*n* | **EOS**}]] ...

- The *cycle* parameter represents the CPU clock cycle where you want an interrupt to occur.

  You can have two types of CPU clock cycles:

  - **Absolute**. To use an absolute clock cycle, your cycle value must represent the actual CPU clock cycle in which you want to simulate an interrupt. For example:

    ```
    12 34 56
    ```

    An interrupt signal is simulated at the 12th, 34th, and 56th CPU clock cycles. Notice that no operation is performed on the clock cycle value; the interrupt occurs exactly as the clock cycle value is written.

Relative. You can also select a clock cycle that is relative to the time at which the last event occurred. For example:

```
12 +34 55
```

In this example, an interrupt signal is simulated at the 12th, 46th (12+34), and 55th CPU clock cycles. A plus sign (+) before a clock cycle adds that value to the clock cycle preceding it. As shown in this example, you can mix both relative and absolute cycle values in your input file.

- The **rpt** {*n* | **EOS**} parameter is optional and represents a repetition value.

  You can have two forms of repetition to simulate interrupts:

  - **Repetition a fixed number of times**. You can format your input file to repeat a particular pattern a fixed number of times. For example:

    ```
    5 (+10 +20) rpt 2
    ```

    The values inside of the parentheses represent the portion that is repeated. Therefore, an interrupt signal is simulated at the 5th, 15th (5+10), 35th (15+20), 45th (35+10), and 65th (45+20) CPU clock cycles.

    Note that *n* is a positive integer value.

  - **Repetition to the end of simulation**. To repeat the same pattern throughout the simulation, add the string EOS to the line. For example:

    ```
    10 (+5 +20) rpt EOS
    ```

    An interrupt signal is simulated at the 10th, 15th (10+5), 35th (15+20), 40th (35+5), 60th (40+20), 65th (60+5), and 85th (65+20) CPU clock cycles, continuing in that pattern until the end of simulation.

After you have created your input file, you can use debugger commands to:

- Connect the interrupt pin to your input file
- List the interrupt pins
- Disconnect an interrupt pin from a file

Use these commands as described below, or use them from the PIN pulldown menu at the top of the debugger display.

To connect your input file to the interrupt pin, use the PINC command:

pinc *pinname, filename*

- The *pinname* parameter identifies the interrupt pin and must be one of the following: EINT1, EINT2, EINT3, or LINT4.

- The *filename* parameter is the name of your input file. Make sure you have set up your input file as described at the beginning of this subsection.

Example 4–1 shows you how to connect an input file with the PINC command.

Example 4–1. Connecting the Input File With the PINC Command

Suppose you want to generate an EINT1 external interrupt at the 12th, 34th, 56th, and 89th clock cycles.

First, create a data file with an arbitrary name, such as myfile:

```
12 34 56 89
```

Then use the PINC command in the pin pulldown menu to connect the input file to the EINT1 pin.

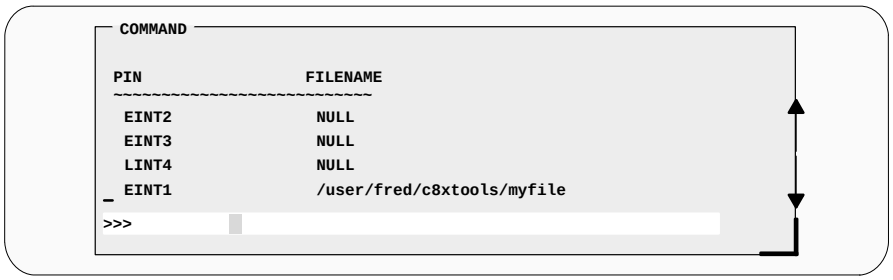
```
pinc  eint1, myfile                    Connects your data file
                                       to the specific interrupt pin
```

This command connects myfile to the EINT1 pin. As a result, the simulator generates an EINT1 interrupt at the 12th, 34th, 56th, and 89th clock cycles.

To verify that your input file is connected to the correct pin, use the PINL command. The syntax for this command is:

pinl

The PINL command displays all of the unconnected pins first, followed by the connected pins. For a connected pin, the debugger displays the name of the pin and the absolute pathname of the file in the COMMAND window.



When you want to connect another file to an interrupt pin, the PINL command is useful for looking up an unconnected pin.

To end the interrupt simulation, you must disconnect the pin. You can do this with the PIND command:

pind *pinname*

The *pinname* parameter identifies the interrupt pin and must be one of the following: EINT1, EINT2, EINT3, or LINT4. The PIND command detaches the file from the interrupt pin. After executing this command, you can connect another file to the same pin.

Support for short-form packet transfers

The MP and PP simulators provide support for short-form packet transfers. For more information, see the TMS320C82 Data Sheet.

Silicon features not supported in the simulator

The following features are *not* supported:

- **Split-read transfer cycles.** Read transfer (memory-to-register) cycles transfer a row from the VRAM memory array into the VRAM serial register (SAM). The MP and PP simulators currently do not support split-read transfer cycles.
- **Split-write transfer cycles.** Write transfer (register-to-memory) cycles transfer data from the SAM into a row of the VRAM memory array. The MP and PP simulators currently do not support split-write transfer cycles.
- **Peripheral device transfers.** Peripheral device transfer mode allows a peripheral device to use the TC's memory controller to read from or write to external 'C8x memory. The MP and PP simulators currently do not support peripheral device transfers.
- **Externally initiated packet transfers.** Externally initiated packet transfers occur when a peripheral device submits a transfer request directly to the TC instead of going through an MP external interrupt. The MP and PP simulators currently do not support externally initiated packet transfers.

4.3 Changes to the MP and PP Compilers

The following subsections detail enhancements made to release 2.00 of the MP and PP compilers.

Source code for the C I/O library

The *TMS320C80 (MVP) Code Generation Tools User's Guide* describes the runtime-support functions included in the C I/O library. Release 2.00 of the compiler now includes source code for these functions. See Table 4–5 for a complete listing of source files. The source files are located in the src directory.

Extending the command line (–@ option)

You can create an extended command line within a temporary file. When you specify that file on the command line with the –@*filename* shell option, the compiler reads the file and interprets it as if it contained part of the command line.

The –@ option allows you to avoid limitations on command line length imposed by the host operating system. Use a # sign at the beginning of a line in the command file to include comments.

Defining a symbol for the assembly module (–ad option)

You can use the –ad option to tell the assembler to define the specified symbol for the assembly module. The syntax for the –ad option is:

–ad*symbol*

Undefining a symbol for the assembly module (–au option)

You can use the –au option to tell the assembler to undefine the specified symbol for the assembly module. The syntax for the –au option is:

–au*symbol*

Generating an auxiliary information file (–b option)

You can create an auxiliary information file that you can refer to for information about stack size and function calls. To generate the auxiliary file, use the –b shell option. The resulting filename is the C source filename with a .aux extension.

Specifying aliased and not aliased variables (–mn option)

By default, previous versions of the MP and PP compilers assumed that variables were not aliased. Release 2.00 of the compiler assumes that variables are aliased by default. If you specify the –ma option, the compiler will still assume aliased variables.

To specify variables as not aliased , use the new –mn option.

For more information on aliased variables, see Sections 1.1.3 and 1.3.3 of the TMS320C80 (MVP) Code Generation Tools User's Guide.

Performing program-level optimization (–o3 and –pm options)

The compiler can now construct a single intermediate representation for an entire program given all of the source files that make up that program. This capability in the parser (enabled with the –o3 and –pm options together) enables more aggressive global optimizations. The compiler is also able to perform intelligent global register allocation. For more information, see Section 5.1, *Using the –o3 Option to Perform File-Level Optimization*, page 5-2.

Altering the level of warning messages (–pw option)

You can determine which levels of warning messages to display by setting the warning message level with the –pw option.

–pw*number*

The number following –pw denotes the level (0,1, or 2). Use Table 4–7 to select the appropriate level.

Table 4–7. Selecting a Level for the –pw Option

If you want to…	**Use option**
Disable all warning messages. This level is useful when you are aware of the condition causing the warning and consider it innocuous.	–pw0
Enable serious warning messages. This is the default.	–pw1
Enable all warning messages.	–pw2

Selecting target and silicon version (–v option)

The MP and PP compilers allow you to specify the target device and the silicon revision. Use the –v option.

–v*target*[**.***version*]
or
–v*version*

- The *target* can be one of the following:
  - **c80** to select the ’C80 device (default)
  - **c82** to select the ’C82 device
- If *target* is **c80**, the *version* can be one of the following:
  - **2** to select pre-production silicon version 2 of the ’C80
  - **3** to select production silicon version 3 of the ’C80 (default)
  - **4** to select production silicon version 4 of the ’C80

    Use version **4** for production silicon version 5 of the ’C80. They are functionally the same; no new features have been added.
- if *target* is **c82**, the *version* can be:
  - **1** to select production silicon version 1 of the ’C82 (default)

Here are some examples of using the –v option:

```
ppcl –vc80.2 test.c      ; Select version 2 of the ’C80
mpcl –vc82 test.c        ; Select version 1 of the ’C82
ppcl test.c              ; Select version 3 of the ’C80
```

If you specify the –z option to enable linking, the program shell (mpcl or ppcl) automatically passes to the linker the –v option that you specify.

The FUNC_EXT_CALLED pragma

When you use the –pm –o3 options, the compiler uses program-level optimization. When you use this type of optimization, the compiler removes any function that is not called, directly or indirectly, by main. You might have C functions that are called by hand-coded assembly instead of main.

The FUNC_EXT_CALLED pragma specifies to the optimizer to keep a C function, *func*, or any other functions that this C function calls. These functions act as entry points into C.

The pragma must appear before any declaration or reference to a function that you want to keep. The syntax of the pragma is:

#pragma FUNC_EXT_CALLED (*func***);**

4.4 Changes to the MP and PP Assemblers

The following subsections detail enhancements made to release 2.00 of the MP and PP assemblers. Section 4.5 details additional enhancements to the PP assembler.

Selecting target and silicon version (–v option)

The MP and PP assemblers allow you to specify the target device and the silicon version when you run the assembler. Use the –v option:

–v*target*[**.***version*]
or
–v*version*

- The *target* can be one of the following:
  - **c80** to select the ’C80 device (default)
  - **c82** to select the ’C82 device
- If *target* is **c80**, the *version* can be one of the following:
  - **2** to select pre-production silicon version 2 of the ’C80
  - **3** to select production silicon version 3 of the ’C80 (default)
  - **4** to select production silicon version 4 of the ’C80

    Use version **4** for production silicon version 5 of the ’C80. They are functionally the same; no new features have been added.
- If *target* is **c82**, the *version* can be:
  - **1** to select production silicon version 1 of the ’C82 (default)

Here are some examples of using the –v option:

```
ppasm –vc80.2 test.s     ; Select version 2 of the ’C80
mpasm –vc82 test.s       ; Select version 1 of the ’C82
ppasm test.s             ; Select version 3 of the ’C80
```

Selecting target and silicon version (.version directive)

The MP and PP assemblers allow you to specify the target device and the silicon version in your code with the .version directive. The syntax for the .version directive is:

.version *target*[**.***version*]
or
.version *version*

- The *target* can be one of the following:
  - **c80** to select the ’C80 device (default)
  - **c82** to select the ’C82 device
- If *target* is **c80**, the *version* can be one of the following:
  - **2** to select pre-production silicon version 2 of the ’C80
  - **3** to select production silicon version 3 of the ’C80 (default)
  - **4** to select production silicon version 4 of the ’C80

    Use version **4** for production silicon version 5 of the ’C80. They are functionally the same; no new features have been added.
- If *target* is **c82**, the *version* can be:
  - **1** to select production silicon version 1 of the ’C82 (default)

4.5 Changes to the PP Assembler

The following subsections detail enhancements made to release 2.00 of the PP assembler. Section 4.4 details additional enhancements to the PP assembler.

Support for saturation ('C82 only)

You can use the t indicator to specify saturation support for the 'C82. You must specify the t indicator following all other sign, split/round multiply, and condition indicators.

An example of multiply saturation is:

```
d2 =t d3*d4 || d5 = ealu(p2: d1+d2+1)
```

An example of ALU saturation is:

```
d2 = d3*d4 || d5 =t ealu(p3: d1+d2+1)
```

An example of both multiply saturation and ALU saturation is:

```
d2 =t d3*d4 || d5 =t ealu(p4: d1+d2+1)
```

For more information on saturation support, see the TMS320C82 Data Sheet.

Support for new 'C8x instructions

Instructions dloop, eloop, and qwait have been added for revision 4 of the 'C80 and for the 'C82. The PP assembler supports these new instructions.

eloop enables the L bit in the PC register that allows loop controllers to be individually enabled with the lctl register. dloop clears the L bit in the PC register thus globally disabling loop control (regardless of lctl settings).

qwait stalls execution of the PP instruction until the comm register's Q bit becomes zero, indicating that any previous packet transfer requests have been completed. When submitting a packet transfer request, a minimum of two delay slots must be inserted between the instruction that sets the P bit of the comm register and the qwait instruction.

For further information on these instructions, see the TMS320C82 Data Sheet.

Clarification of the explicit EALU function syntax

The following general equation describes all of the 256 arithmetic functions supported by the PP's three-input ALU now supported by the PP assembler:

```
dst1 = (A & f1(B,C)) + f2(B,C) [+1]
```

f1(B,C) and f2(B,C), referred to as subfunctions, can be any of the possible functions of B and C shown in Table 4–8. f1(B,C) and f2(B,C) are independent of each other. You can add a carry-in to any equation, except where noted. The +1 carry-in transforms any logical NOT (~) expression to a unary minus (–) since ~x = –x – 1.

Table 4–8 lists the 18 possible functions of B and C. This table replaces Table 4–2, *The 16 Possible Functions of B and C (Assuming Carry-In = 0)*, and Table 8–27, *Possible f1(B,C) or F2(B,C) Functions*, in the 1995 *TMS320C80 (MVP) Parallel Processor User's Guide*.

Table 4–8. The 18 Possible Functions of B and C

f1 Code	f2 Code	Subfunction	Common Use
00	00	0	Do not place an explicit zero in the equation. Eliminate the zeroed subexpression from the equation.
AA	FF	–1	All 1s or –1.
88	CC	B	Return B.
22	33	~B	Return NOT B. If the +1 carry-in is included, negate B.
A0	F0	C	Return C.
0A	0F	~C	Return NOT C. If the +1 carry-in is included, negate C.
80	C0	B&C	Force bits in B to 0 where C is 0.
2A	3F	~(B&C)	Force bits in B to 0 where C is 0 and invert. If the +1 carry-in is included, force bits in B to 0 where C is 0 and negate.
A8	FC	B | C	Force bits in B to 1 where C is 1.
02	03	~(B | C)	Force bits in B to 1 where C is 1 and invert. If the +1 carry-in is included, force bits in B to 1 where C is 1 and negate.
08	0C	B&~C	Force bits in B to 0 where C is 1.
A2	F3	~(B&~C)	Force bits in B to 0 where C is 1 and invert. If the +1 carry-in is included, force bits in B to 0 where C is 1 and negate.
8A	CF	B | ~C	Force bits in B to 1 where C is 0.
20	30	~(B | ~C)	Force bits in B to 1 where C is 0 and invert. If the +1 carry-in is included, force bits in B to 1 where C is 0 and negate.
28	3C	(B&~C) | ((~B) &C)	Choose B if C = all 0s, or NOT B if C = all 1s.
N/A	3C	(B&~C) | ((–B) &C)†	Choose B if C = all 0s, or –B if C = all 1s.
82	C3	(B&C) | ((~B) &~C)	Choose B if C = all 1s, or NOT B if C = all 0s.
N/A	C3	(B&C) | ((–B) &~C)†	Choose B if C = all 1s, or –B if C = all 0s.

† This subfunction can be used only with f2. A +1 carry-in cannot be added to equations using this syntax.

An example using the explicit EALU format

To use the explicit EALU format, you must do the following:

- **Determine the inputs available for ports A, B, and C.** For more information, see subsection A.3.1, *Data Unit Opcode Format A: Six-Operand*, in the 1995 *TMS320C80 (MVP) Parallel Processor User's Guide.*
- **Determine the src and dst registers.** To determine the src and dst registers, you must select a parallel transfer format. The formats are described in Section A.4, *Parallel Transfer Opcode Formats*, in the 1995 *TMS320C80 (MVP) Parallel Processor User's Guide.*

  Parallel transfer formats 6 through 10 allow dst1 to be any valid PP register because these formats contain an Adstbnk field. They also contain an As1bank field, which allows src1 to be any valid PP register. If you use any other parallel transfer format, all source and destination registers must be D registers.

If no parallel transfer is necessary, the assembler uses one of the parallel transfer formats 6 or 10 so that src1 and dst1 can be any valid PP registers.

For example, consider the following instruction:

```
dst2=src3*src4 ||
   dst1=ealu(label: (A&(B&C))+((B&C)|((–B)&~C)) )
```

This is a valid multiply with a parallel extended ALU operation (MPY||EALU) where f1=B&C and f2=(B&C)|((–B)&~C). To determine possible port assignments, see page PP: A-9 of the 1995 *TMS320C80 (MVP) Parallel Processor User's Guide*:

- The A port input must be src2.
- The B port input must be src1 rotated left by the 5-bit immediate barrel rotate amount (DBR field) stored in the d0 register.
- The C port can be either a mask generated from the 5-bit DBR field of the d0 register or from the expanded mf register (specified by @mf).

  For this example, assume the 5-bit mask for the C port input, with an immediate value of 3.

Since the mask shares the 5-bit immediate value with the barrel rotate, both must contain the value 3.

Given this information, the instruction is now:

```
dst2=src3*src4 ||
   dst1=ealu(label: (src2&((src1\\3)&%3))+
   (((src1\\3)&%3)|((–(src1\\3))&~%3)) )
```

Since there is no parallel transfer, this example is not limited to a specific parallel transfer format. Therefore, src1 and dst1 can be any PP register from Table PP:7–1, *The Register Codes*, in the 1995 *TMS320C80 (MVP) Parallel Processor User's Guide*.

For this example, assign the sr register to src1 and the le0 register to dst1.

Recall that all other registers must be D registers; for this example, assign the registers as follows:

- dst2 = d2
- src2 = d5
- src3 = d3
- src4 = d4

Here is the final, executable instruction:

```
d2=d3*d4 ||
   le0=ealu(label: (d5&((sr\\3)&%3))+
   (((sr\\3)&%3)|((-(sr\\3))&~%3)) )
```

4.6 Changes to the Linker

The following subsections detail enhancements made to release 2.00 of the linker.

Invoking the 'C82 version of the linker (–vc82 option)

The linker now supports the 'C82. To simulate the 'C82, you must specify the –vc82 option when you invoke the linker directly. If you invoke the linker from the program shell (mpcl or ppcl) with the –z option, the correct version is automatically passed to the linker.

Inserts nop instead of zero

The linker now inserts nops when alignment forces gaps in executable code. In the previous version, the linker inserted zeros.

Support for subsections

Subsections are smaller sections within larger sections. Like sections, subsections can be manipulated by the linker. Subsections give you tighter control of the memory map. You can create subsections by using the .sect or .usect directive. The syntax for a subsection name is:

section name:*subsection name*

A subsection is identified by the base section name followed by a colon, then the name of the subsection. A subsection can be allocated separately or grouped with other sections using the same base name. For example, to create a subsection called _func within the .text section enter the following:

```
.sect ".text:_func"
```

_func could be allocated separately or within all of the .text sections.

Addition of warning messages

The following warning messages have been added to assist you when you are linking your code:

`Section "`*`section name`*`" user requested `*`alignment value`*` run alignment is overridden by `*`filename`*` input section requirements to `*`alignment value`*`.`

Description The user requested alignment differs from the alignment specified by the input section. For example, if your assembly file specifies a .align 256 but the code is linked in a section with 16-byte alignment, this warning would be issued to inform you of potentially unnoticed space used for padding.

`Section "`*`section name`*`" `*`alignment value`*` run alignment is different than the [`*`alignment value`*`] load alignment.`

Description The load alignment and the run alignment differ. For example, when copying sections from load-space to run-space certain assumptions are made concerning the consistency of section offsets. This warning will be issued to notify you that the offsets may be different.

4.7 Changes to the Multitasking Executive

The following changes have been made to release 2.00 of the multitasking executive:

- The task.h include file, located in the exec/lib directory, includes definitions for MAX_NUM_PORTS and MAX_NUM_SEMAS. These definitions set the limits for the maximum number of ports and semaphores. In this release, the values for MAX_NUM_PORTS and MAX_NUM_SEMAS have increased from 32 to 64.
- The multitasking executive software now includes little-endian libraries.

4.8 Tool Defects and Suggested Work-Arounds

Since the last release, Texas Instruments has identified and fixed several defects in the TMS3208x software development tools. However, some defects that have been identified are still not resolved. These unresolved defects are listed in the following subsections. Whenever possible, a work-around has been provided also.

You should read this section before you start using the tools to help you avoid known defects. You should also refer to this section while you are using the tools if you think you have found a defect to determine whether or not Texas Instruments is aware of the problem and to see if a work-around has been provided.

C compiler defect

Defect When you cast a function call to an integer type and define the function to return a pointer, an integer value is returned instead of a pointer value. For example, when this code is compiled an undefined value will be assigned to int_return and the function return value will be ignored:

```
char *func_return_pointer();
main()
{
int int_return;
int_return = (int)func_return_pointer();
}
char *func_return_pointer()
{
return (char*)0x10000000;
}
```

Work-around Copy the function return value to a variable and use the variable in place of the function call. The code example above could be rewritten as follows:

```
char *func_return_pointer();
main()
{
char *char_tmp;
int int_return;
char_tmp = func_return_pointer();
int_return = (int)char_tmp;
}
char *func_return_pointer()
{
return (char*)0x10000000;
}
```

Assembler defect

Defect It is not always necessary to utilize both subfunctions in an EALU expression. (Refer to the *Clarification of the explicit EALU function syntax* subsection on page 4-23 for more information about subfunctions.) There are two ways to do this: one way is to omit the subfunction; the other way is to use a 0 (zero) in place of the subfunction. The assembler might fail if you use zero; the recommended method is to omit the subfunction.

Work-around Omit the subfunction instead of inserting zero.

Debugger defects

Defect Directly before the execution of main(), the color of the variables in the WATCH window indicates that the uninitialized values are valid.

Work-around Ignore colors in the WATCH window directly before the execution of main().

Defect (Simulator only) When packet transfers or external memory accesses occur, the clock cycles recorded by TCOUNT are inaccurate.

Work-around A work-around is not available at this time.

Defect The debugger does not handle large, multidimensional arrays properly. Here is an example:

If you have the following global variable declaration in a C file:

```
#pragma DATA_SECTION( pp_data, ”mysection” )

unsigned char pp_data[ 4 ][ 0x1c200 ];
```

And you have the following statements in your linker command file:

```
MEMORY
{
MYMEM:  0 = 0x02600000    l = 0x00100000
}
SECTIONS
{
mysection > MYMEM
}
```

The linker yields the following addresses:

Expression	Address
`&( pp_data[ 0 ][ 0 ] )`	`0x02600000`
`&( pp_data[ 1 ][ 0 ] )`	`0x0261c200`
`&( pp_data[ 2 ][ 0 ] )`	`0x02638400`
`&( pp_data[ 3 ][ 0 ] )`	`0x02654600`

If you enter these expressions in the command window of the debugger, the debugger gives you the incorrect addresses:

Expression	Address
`? &( pp_data[ 0 ][ 0 ] )`	`0x02600000`
`? &( pp_data[ 1 ][ 0 ] )`	`0x0260c200`
`? &( pp_data[ 2 ][ 0 ] )`	`0x02618400`
`? &( pp_data[ 3 ][ 0 ] )`	`0x02624600`

The debugger generates the wrong addresses, because it uses only the 16 least significant bits of the second dimension size to calculate the addresses.

Work-around Declare the largest dimension of the array first. The declaration in the example above could be rewritten as follows:

```
unsigned char pp_data[ 0x1c200 ][ 4 ];
```

Defect (ppsim only) When the halt bit in the halt override register is set and you perform two consecutive reset commands, the debugger generates the following error message:

```
Attempt to fill cache from crossbar memory
```

Work-around A work-around is not available at this time. Avoid consecutive reset commands.

Defect (ppsim only) The simulator does not show the correct disassembly for many EALU functions; however, execution is performed correctly.

Work-around A work-around is not available at this time.

PDM defect

Defect When the PP simulator is spawned before the MP simulator, the behavior of the simulators is unpredictable.

Work-around Make sure the MP simulator is spawned before the other simulators.

Chapter 5

Optimizing Your Program Code

This chapter contains useful tips on performing file-level and program-level compiler optimizations.

Topic **Page**

5.1 Using the –o3 Option to Perform File-Level Optimization

The –o3 option instructs the compiler to perform file-level optimization. You can use the –o3 option alone to perform general file-level optimization, or you can combine it with other options to perform more specific optimization. The following options interact with –o3 to perform the indicated optimization:

If you ...	Use option	See ...
Want to combine source files, the optimizer can perform program-level optimization	–pm	Section 5.2, *Using the –pm and –o3 Options to Perform Program-Level Optimization*
Have files that redeclare standard library functions, the optimizer can perform file-level optimization	–ol	The *Controlling file-level optimizations* subsection on page 5-2
Want to create an optimization information file	–on	The *Creating an optimization information file* subsection on page 5-3

Controlling file-level optimizations

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

Table 5–1. Selecting a Level for the –ol Option

If your source file...	Use this option
Declares a function with the same name as a standard library function	–ol0
Contains but does not alter functions declared in the standard library	–ol1
Does not alter standard library functions, but you used the –ol0 or –ol1 option in a command file or an environment variable, and now you want to restore the default behavior of the optimizer	–ol2

Creating an optimization information file

When you invoke the optimizer with the –o3 option, you can use the –on option to create an optimization information file that you can read. The number following the –on denotes the level (0, 1, or 2). The resulting file has an .nfo extension. Use Table 5–2 to select the appropriate level to append to the –on option.

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

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

5.2 Using the –pm and –o3 Options to Perform Program-Level Optimization

If you want to combine source files, the optimizer can perform program-level optimization. Use the –pm option with the –o3 option to specify program-level optimization.

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

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

Optimization considerations when mixing C and assembly

If you have any assembly functions in your program, you need to exercise caution when using the –pm option. The compiler recognizes only the C source code and not any assembly code that might be present. Because the compiler does not recognize the assembly code calls and variable modifications to C functions, the –pm option optimizes out those C functions. One way to keep these functions is to place the FUNC_EXT_CALLED pragma (see *The FUNC_EXT_CALLED pragma* subsection on page 4-19) before any declaration or reference to a function that you want to keep.

Another approach you can take when you use assembly functions in your program is to use the –op option with the –pm and –o3 options (see the *Controlling program-level optimization* subsection on page 5-6).

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

If any of the following situations apply to your application, use the suggested solution:

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

Solution Compile with –pm –o3 –op2 to tell the compiler that outside functions do not call C functions or modify C variables. See the *Controlling program-level optimization* subsection on page 5-6 for information about the –op2 option.

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

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

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

- Compile with –pm –o3 –op1.
- Add the volatile keyword to those variables that may be modified by the assembly functions and compile with –pm –o3 –op2. For more information on the volatile keyword, see the *C Compiler Description* chapter of the *TMS320C80 (MVP) Code Generation Tools User's Guide*.

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

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

- Apply the FUNC_EXT_CALLED pragma to all of the entry-point functions called from the assembly language interrupts, and then compile with –pm –o3 –op2. This gives the best optimization. *Ensure that you use the pragma with all of the entry-point functions. If you do not, the compiler removes the entry-point functions that are not preceded by the FUNC_EXT_CALLED pragma.*
- Compile with –pm –o3 –op3. Because you do not use the FUNC_EXT_CALLED pragma, you must use the –op3 option, which is less aggressive than the –op2 option, and your optimization may not be as effective.

See the *Controlling program-level optimization* subsection for information about the –op option. To see which program-level optimizations the compiler is applying, use the –on2 option to generate an information file. See the *Creating an optimization information file* subsection on page 5-3 for more information.

Controlling program-level optimization

You can control program-level optimization, which you invoke with –pm –o3, by using the –op option. Specifically, the –op option indicates if functions in other modules can call a module's external functions or modify a module's external variables. The number following –op indicates the level you set for the module that you are allowing to be called or modified. The –o3 option combines this information with its own file-level analysis to decide whether to treat this module's external function and variable declarations as if they had been declared static. Use Table 5–3 to select the appropriate level to append to the –op option.

Table 5–3. Selecting a Level for the –op Option

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

In certain circumstances, the compiler reverts to a different –op level from the one you specified or disables program-level optimization altogether. Table 5–4 lists the combinations of –op levels and conditions that cause the compiler to revert to other –op levels.

Table 5–4. Special Considerations When Using the –op Option

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

In some situations when you use –pm and –o3, you *must* use an –op option or the FUNC_EXT_CALLED pragma. See the *Optimization considerations when mixing C and assembly* subsection on page 5-4 for information about these situations.

Index

D

E

F

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z