TMS340 Family
C Source Debugger
User's Guide

PRINTED WITH
SOY INK ™

ti TEXAS
INSTRUMENTS

Printed on Recycled Paper

TEXAS INSTRUMENTS

TMS340 Family C Source Debugger Reference Card

Phone Numbers

TI Customer Response Center (CRC) Hotline: (800) 336–5236

Graphics Hotline: (713) 274–2340

Debugging TIGA Applications

Before invoking the debugger, install the TIGA communication driver:

Development Board:	**TIGACD / D2**
	Execute TIGACOM on the target system.
	Execute DEBUGCOM on the host system.
Emulator:	**TIGACD / D1**

Invoking the Debugger

Development Board:	**db340**	[*filename*]	[*–options*]
Emulator:	**db340emu**	[*filename*]	[*–options*]

Debugger Options

Option	Description
–b[bbbb]	Screen size options.
	Option Chars./Lines Notes
	none 80 X 25 Default display
	–b 80 X 43† Use any EGA
	80 X 50‡ or VGA card
	–bb 120 X 43
	–bbb 132 X 43 Supported on a
	–bbbb 80 X 60 Video Seven
	–bbbbb 100 X 60 VEGA Deluxe card
–i *pathname*	Identifies additional directories that contain source files.
–mc	Tells the debugger to provide '34082 support.
–mf	Tells the debugger to expect IEEE floating–point format.
–mi	Tells the debugger not to initialize the program counter (PC) or the stack pointer (SP). ††

†   EGA card
‡   VGA card
†† Development Board Version only

Debugger Options

Option	Description
–p *port address*	**Emulator only.** If you used nondefault switch settings, you must use –p**.**

Switch 1	Switch 2	Option
on	on	–p 220
on	off	*none needed*
off	on	–p 300
off	off	–p 3E0

Option	Description
–s	Tells the debugger to load *filename*'s symbol table only.
–t *filename*	Specifies an initialization command file to be used instead of init.cmd.
–v	Loads only global symbols; later, local symbols are loaded as needed. Affects **all** loads.
–x	Ignores options supplied with D_OP-TIONS.

Summary of Debugger Commands

Command	Description
? *expression*	Evaluate *expression*
addr *address* **addr** *function name*	Display code at *address* or *function name*
alais [*aliasname* **[''***command string***'']]**	Associates one or more debugger commands with a single *alias name*.
asm	Switch to assembly mode
ba *address*	Set breakpoint at *address*
bd *address*	Delete breakpoint at *address*
bl	List all breakpoints
border [*active*] [,*inactive*] [,*resize*]	Change window border style
br	Clear all breakpoints
c	Switch to C/auto mode
calls	Open the CALLS window
cd *directory name* **chdir** *directory name*	Change current directory
cls	Clear COMMAND display
cnext [*expression*]	Single-step C code, step over functions
color *area, attr$_1$* [,*attr$_2$* [,*attr$_3$* [,*attr$_4$*]]]	Change screen colors, delayed update
cstep [*expression*]	Single-step C code
dasm *address* **dasm** *function name*	Display disassembly at *address* or *function name*
dir [*directory*]	Show contents of *directory*
disp *expression*	Open DISPlay window

Summary of Debugger Commands

Command	Description
eval *expression* **e** *expression*	Evaluate *expression*
file *filename*	Display text file
fill *address, length, data*	Fill memory
fpuregs	Opens the FPU window to display the '34082 registers.
func *function name* **func** *address*	Display function
go [*address*]	Run to *address*
halt[†]	Halt target system
ioregs	Opens the I/O window to display the registers.
load *object filename*	Load object file
ma *address, length, type*	Add block to memory map
map {**on** | **off**}	Enable / disable memory mapping
md *address*	Delete block from memory map
mem *expression*	Display memory at *expression*
mix	Switch to mixed mode
ml	List blocks in memory map
mod [*module identifier*]	TIGA module identifier
move [*X, Y* [, *width, length*]]	Move active window
mr	Reset memory map
ms *addresslength, filename*	Saves the parameter values in a block of memory to a system file.
next [*expression*]	Single-step disassembly, step over functions
prompt *new prompt*	Change the prompt
quit	Exit the debugger
reload *object filename*	Load object file without symbol table
reset	Reset target system (emulator) or reload gspmon (development boards)
restart, rest	Reset PC to program entry point
return, ret	Return to function's caller
run [*expression*]	Run program
runb[†]	Benchmark code
runf[†]	Run free from target
scolor *area, attr$_1$* [, *attr$_2$* [, *attr$_3$* [, *attr$_4$*]]]	Change screen colors, immediate update
sconfig [*filename*]	Load saved screen configuration
setf [*datatype, display format***]**	Changes the display format for a specific datatype.

† Emulator only

Summary of Debugger Commands

Command	Description
size [*width*, *length*]	Size active window
sload *object filename*	Load object file's symbol table
sound on | off	Beeps every time an error message is displayed.
ssave [*filename*]	Save current screen configuration
step [*expression*]	Single-step disassembly
system [*operating–system command* [*arg*]]	Allows you to enter operating–system commands without exiting the debugger.
take *filename* [*, flag*]	Execute batch file
tba *function name*	Set a tentative breakpoint on a TIGA module
tbd *breakpoint index*	Clear a tentative breakpoint
unalias *aliasname*	Deletes an alias and its definition.
use *directory name*	Use an additional directory
wa *expression*[*, label*]	Add item to WATCH window
wd *index number*	Delete item from WATCH window
whatis *symbol*	Query type of *symbol*
win *WINDOW NAME*	Make WINDOW active
wr	Reset WATCH window
zoom	Makes the active window as large as possible.

† Emulator only

Border Styles
(BORDER Command)

Index	Style
0	Double-lined box
1	Single-lined box
2	Solid 1/2-tone top, double-lined sides/bottom
3	Solid 1/4-tone top, double-lined sides/bottom
4	Solid box, thin border
5	Solid box, heavy sides, thin top/bottom
6	Solid box, heavy borders
7	Solid 1/2-tone box
8	Solid 1/4-tone box

Colors and Attributes
(COLOR/SCOLOR Commands)

black	blue	green	cyan
red	magenta	yellow	white
bright		blink	

Area Names
(COLOR/SCOLOR Commands)

menu_bar	menu_border	menu_entry	menu_cmd
menu_hilite	menu_hicmd	win_border	win_hiborder
win_resize	field_text	field_hilite	field_edit
field_label	field_error	cmd_prompt	cmd_input
cmd_cursor	cmd_echo	asm_data	asm_cdata
asm_label	asm_clabel	background	blanks
error_msg	file_line	file_eof	file_text
file_brk	file_pc	file_pc_brk	

Window Size and Position Limits
(SIZE and MOVE Commands)

Screen size	Option	Valid widths	Valid lengths	Valid X pos.	Valid Y pos.
80 X 25	none	4–80	3–24	0–76	1–22
80 X 43[†] 80 X 50[‡]	–b	4–80	3–42 3–49	0–76	1–40 1–47
120 X 43	–bb	4–120	3–42	0–116	1–40
132 X 43	–bbb	4–132	3–42	0–128	1–40
80 X 60	–bbbb	4–80	3–59	0–76	1–57
100 X 60	–bbbbb	4–100	3–59	0–106	1–57

[†] EGA card
[‡] VGA card

Memory Types (MA Command)

To identify this kind of memory	Use this keyword as the *type* parameter
read-only memory	**R**, **ROM**, or **READONLY**
write-only memory	**W**, **WOM**, or **WRITEONLY**
read/write memory	**RW** or **RAM**
no-access memory	*PROTECT*

Switching Modes

| To do this	Use this function key
Switch debugging modes in this order:	F3

auto ⟶ assembly ⟶ mixed

Editing Text on the Command Line

To do this	Use these function keys
Enter the current command (if you press the return key in the middle of text, the debugger truncates the input text at the point where you press this key)	[↵]
Move back over text without erasing characters	[CTRL] [H] or [BACK SPACE]
Move forward through text without erasing characters	[CTRL] [L]
Move back over text while erasing characters	[DELETE]
Move forward through text while erasing characters	[SPACE]
Insert text into the characters that are already on the command line	[INSERT]

Using the Command History

To do this	Use these function keys
Move backward, one command at a time, through the command history	[TAB]
Move forward, one command at a time, through the command history	[SHIFT] [TAB]

Halting or Escaping From an Action

To do this	Use this function key
❑ Halt program execution ❑ Close a pulldown menu ❑ Undo an edit of the active field in a data-display window (pressing this key leaves the field unchanged) ❑ Halt the display of a long list of data in the COMMAND window display area	[ESC]

Displaying Pulldown Menus

To do this	Use these function keys
Display the Load menu	`ALT` `L`
Display the Break menu	`ALT` `B`
Display the Watch menu	`ALT` `W`
Display the Memory menu	`ALT` `M`
Display the Color menu	`ALT` `C`
Display the MoDe menu	`ALT` `D`
Display an adjacent menu	`←` or `→`
Execute any of the choices from a displayed pulldown menu	Press the high-lighted letter corresponding to your choice

Running Code

To do this	Use these function keys
Run code from the current PC (equivalent to the RUN command without an *expression* parameter)	`F5`
Single-step code from the current PC (equivalent to the STEP command without an *expression* parameter)	`F8`
Single-step code from the current PC; step over function calls (equivalent to the NEXT command without an *expression* parameter)	`F10`

Selecting or Closing a Window

To do this	Use these function keys
Select the active window (pressing this key makes each window active in turn; stop pressing the key when the desired window becomes active)	`F6`
Close the CALLS or DISP window (the window must be active before you can close it)	`F4`
Repeat the last command	`F2`

Editing Data or Selecting the Active Field

To do this	Use this function key
❑ *FILE or DISASSEMBLY window:* Set or clear a breakpoint	`F9`
❑ *CALLS window:* Display the source to a listed function	
❑ *Any data-display window:* Edit the contents of the current field	
❑ *DISP window:* Open an additional DISP window to display a member that is an array, structure, or pointer	

Moving or Sizing a Window

Enter the MOVE or SIZE command without parameters, then use the arrow keys:

To do this	Use these function keys
❑ Move the window down one line	↓
❑ Make the window one line longer	
❑ Move the window up one line	↑
❑ Make the window one line shorter	
❑ Move the window left one character position	←
❑ Make the window one character narrower	
❑ Move the window right one character position	→
❑ Make the window one character wider	

Scrolling the Active Window's Contents

To do this	Use these function keys
Scroll up through the window contents, one window length at a time	PAGE UP
Scroll down through the window contents, one window length at a time	PAGE DOWN
Move the field cursor up, one line at a time	↑
Move the field cursor down, one line at a time	↓
❑ *FILE window only:* Scroll left, 8 characters at a time	←
❑ *Other windows:* Move the field cursor left 1 field; at the first field on a line, wrap back to the last fully displayed field on the previous line	
❑ *FILE window only:* Scroll right, 8 characters at a time	→
❑ *Other windows:* Move the field cursor right 1 field; at the last field on a line, wrap around to the first field on the next line	
FILE window only: Adjust the window's contents so that the first line of the text file is at the top of the window	HOME
FILE window only: Adjust the window's contents so that the last line of the text file is at the bottom of the window	END
DISP windows only: Scroll up through an array of structures	CTRL PAGE UP
DISP windows only: Scroll down through an array of structures	CTRL PAGE DOWN

8

Read This First

What Is This Book About?

This book tells you how to use the the '340 family C source debugger with these debugging tools:

❏ The '34020 emulator

❏ '34010- and '34020-based development boards that use TIGA communication software (version 2.05 or later). This includes the Texas Instruments '34010 TIGA development board, '34020 software development board, and a number of third-party '340 development boards.

How to Use This Book

The goal of this book is to help you install the C source debugger and learn how to use it. This book is divided into three distinct parts:

❏ **Part I: Hands-On Information** is presented first so that you can start using your debugger the same day that you receive it.

■ There are two versions of the debugger—one for development boards and one for the emulator—and two sets of installation instructions (Chapters 2 and 3). *It is very important to use the correct installation*—Chapter 1 will help you to select the appropriate installation chapter.

■ Chapter 4 is a tutorial that introduces you to many of the debugger features.

❏ **Part II: Debugger Description** contains detailed information about using the debugger.

■ Chapter 5 is analogous to a traditional manual introduction. It lists the key features of the debugger, describes additional '340 software tools, and tells you how to prepare a '340 program for debugging.

■ The remaining chapters in Part II detail the individual topics that are introduced in the tutorial. For example, Chapter 6 describes all of the debugger's windows and tells you how to move them and size them; Chapter 7 describes everything you need to know about entering commands.

❏ **Part III: Reference Material** provides supplementary information.

■ Chapter 13 provides a complete reference to all the tasks introduced in Parts I and II. This includes a functional and an alphabetical reference of the debugger commands and a topical reference of function key actions.

■ Chapter 14 provides information about C expressions. The debugger commands are powerful because they accept C expressions as parameters; however, the debugger can also be used to debug assembly language programs. The information about C expressions will aid assembly language programmers who are unfamiliar with C.

■ Part III also includes a glossary and an index.

The way you use this book should depend on your experience with similar products. As with any book, it would be best for you to begin on page 1 and read to the end. Because most people don't read technical manuals from cover to cover, here are some suggestions about what you should read.

❏ If you have used TI development tools or other debuggers before, then you may want to:

■ Read Chapter 1 to determine which version of the debugger you should install.
■ Use the appropriate installation chapter, as directed in Chapter 1.
■ Complete the tutorial in Chapter 4.
■ Read the alphabetical command reference in Chapter 13.

❏ If this is the first time that you have used a debugger or similar tool, then you may want to:

■ Read Chapter 1 to determine which version of the debugger you should install.
■ Use the appropriate installation chapter, as directed in Chapter 1.
■ Complete the tutorial in Chapter 4.
■ Read all of the chapters in Part II.

Notational Conventions

This document uses the following conventions:

❏ The C source debugger has a very flexible command-entry system; there are usually a variety of ways to perform any specific action. For example, you may be able to perform the same action by typing in a command, using the mouse, or using a function key. This document uses three symbols to identify the methods that you can use to perform an action:

Symbol   Description

Identifies an action that you perform by using the mouse.

Identifies an action that you perform by using function keys.

Identifies an action that you perform by typing in a command.

❏ The following symbols identify mouse actions. For simplicity, these symbols represent a mouse with two buttons; however, you can use a mouse with only one button or a mouse with more than two buttons.

Symbol   Action

 K   *Point.* Without pressing a mouse button, move the mouse to point the cursor at a window or field on the display. (Note that the mouse cursor displayed on the screen is not shaped like an arrow; it's shaped like a block.)

▮   *Press and hold.* Press a mouse button. If your mouse has only one button, press it. If your mouse has more than one button, press the left button.

          Release. Release the mouse button that you pressed.

✕   *Click.* Press a mouse button and, without moving the mouse, release the button.

▐   *Press, hold, and move.* While pressing the left mouse button, move the mouse.

❏ Debugger commands are not case sensitive; you can enter them in lowercase, uppercase, or a combination. To emphasize this fact,

commands are shown throughout this user's guide in both uppercase and lowercase.

❏ The debugger recognizes standard C numeric formats. Hexadecimal numbers must be prefixed with **0x.** Octal numbers must be prefixed with **0**. Decimal numbers are not prefixed.

❏ Program listings and examples, interactive displays, and window contents are shown in a special font. Some examples use a **bold version** to identify code, commands, or portions of an example that *you* enter. Here is an example:

Enter	Result displayed in the COMMAND window
whatis giant	`struct zzz giant[100];`

whatis xxx	`struct xxx    {`
	`    int a;`
	`    int b;`
	`    int c;`
	`    int f1 : 2;`
	`    int f2 : 4;`
	`    struct xxx *f3;`
	`    int f4[10];`
	`}`

In this example, the left column identifies debugger commands that you type in. The right column identifies the result that the debugger displays in the COMMAND window display area.

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

wa *expression* [, *label*]

wa is the command. This command has two parameters, indicated by *expression* and *label*. The first parameter must be an actual C expression; the second parameter, which can be any string of characters, is optional.

❏ 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 an optional parameter:

run [*expression*]

The RUN command has one parameter, *expression*, which is optional.

Related Documentation From Texas Instruments

The following books describe the TMS34010, TMS34020, and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Customer Response Center (CRC) at (800) 336–5236. When ordering, please identify the book by its title and literature number.

TMS34010 TIGA Development Board User's Guide (literature number SPVU031) provides an in-depth description of TIGA development board operation.

TMS34020 Software Development Board User's Guide (literature number SPVU034) provides an in-depth description of the software development board operation.

TMS34020 Emulator Installation Guide (literature number SPVU032) provides an in-depth description of emulator board operation.

TMS34010 User's Guide (literature number SPVU001) discusses hardware aspects of the '34010, such as pin functions, architecture, stack operation, interfaces, and instruction set.

TMS340 Graphics Library User's Guide (literature number SPVU027) describes the graphics operations library that is available for a '340-based graphics system.

TIGA Interface User's Guide (literature number SPVU015) describes the architecture of the TIGA (Texas Instruments Graphics Architecture) software interface between a host processor and a '340 graphics processor, which includes the applications interface, communications driver, and graphics manager.

TMS34020 User's Guide (literature number SPVU019) describes hardware aspects of the '34020, such as pin functions, architecture, stack operation, interfaces, and instruction set.

TMS340 Family Assembler Support for the TMS34082 (literature number SPVU029) summarizes the '34082 instruction set.

TMS340 Family Code-Generation Tools User's Guide (literature number SPVU020) describes the '340 C compiler, assembler, linker, archiver, and auxiliary tools that are available for developing '34010 or '34020 code.

Pixel Perspectives is a quarterly newsletter, published by the Computer Video Products group of Texas Instruments Incorporated. This newsletter describes new products, discusses support for existing products, and identifies new documentation releases.

If you are an assembly language programmer and would like more information about C or C expressions, you may find this book useful:

The C Programming Language (second edition, 1988), by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice–Hall, Englewood Cliffs, New Jersey.

If You Need Assistance. . .

If you want to. . .	Do this. . .
Request more information about Texas Instruments computer video products	Call the CRC[†] hotline: **(800) 336–5236**
	Or write to Texas Instruments Incorporated Market Communications Manager, MS 736 P.O. Box 1443 Houston, Texas 77251–1443
Order Texas Instruments documentation	Call the CRC[†] hotline: **(800) 336–5236**
Ask questions about product operation or report suspected problems	Call the graphics hotline: **(713) 274–2340**
Report mistakes in this document or any other TI documentation	Send your comments to Texas Instruments Incorporated Technical Publications, MS 702 P.O. Box 1443 Houston, Texas 77251–1443

[†] Texas Instruments Customer Response Center; the number is toll free in the U.S. and Canada.

Trademarks

TIGA is a trademark of Texas Instruments Incorporated.

PC-DOS is a trademark of International Business Machines.

MS-DOS and Windows are trademarks of Microsoft Corporation.

VEGA Deluxe is a trademark of Video Seven Incorporated.

Contents

Part I: Hands-On Information

Part II: Debugger Description

7   Entering and Using Commands 7-1

Describes the rules for entering commands from the command line, tells you how to use the pulldown menus and dialog boxes (for entering parameter values), describes general information about entering commands from batch files, and describes the use of DOS-like system commands.

8   Defining a Memory Map 8-1

Contains instructions for setting up a memory map that will enable the debugger to correctly access target memory. Also includes hints about using batch files.

12 Using Breakpoints

Describes the use of software breakpoints to halt code execution.

13 Customizing the Debugger Display

Contains information about the commands that you can use for customizing the display, and identifies the display areas that you can modify.

Part III: Reference Material

14 Summary of Commands and Special Keys 14-1

Provides a functional summary of the debugger commands and function keys; also provides a complete alphabetical summary of all debugger commands.

15 Basic Information About C Expressions 15-1

Many of the debugger commands accept C expressions as parameters. This chapter provides general information about the rules governing C expressions and describes specific implementation features related to using C expressions as command parameters.

Figures

Tables

Identifying the Correct Installation for Your Development System

The '340 C source debugger is a software interface for '340 debugging systems. There are two versions of the debugger:

The **development board version** of the C source debugger works with '340-based PC development boards such as the '34010 TIGA development board or the '34020 software development board. It also works with third-party boards that use a TIGA communication driver.

The **emulator version** of the C source debugger works with the '34020 emulator.

Both versions of the debugger operate almost identically. However, the executable files that invoke them are very different. The development board version will not work with the emulator, and vice versa. Be sure to install the correct version of the debugger for your environment.

This chapter describes how these two versions of the debugger can be used in various environments. It also tells you which installation chapter to use to make sure that you install the correct version of the debugger for your environment.

An emulator and development board are not usually used in the same system. However, when they are, here's some extra information that you'll need.

1.1 Using the Development Board Version of the Debugger

Figure 1–1 shows the key hardware and software items that you'll need for using the development board version of the C source debugger.

Figure 1–1. Key Items in a '340-Based Development System
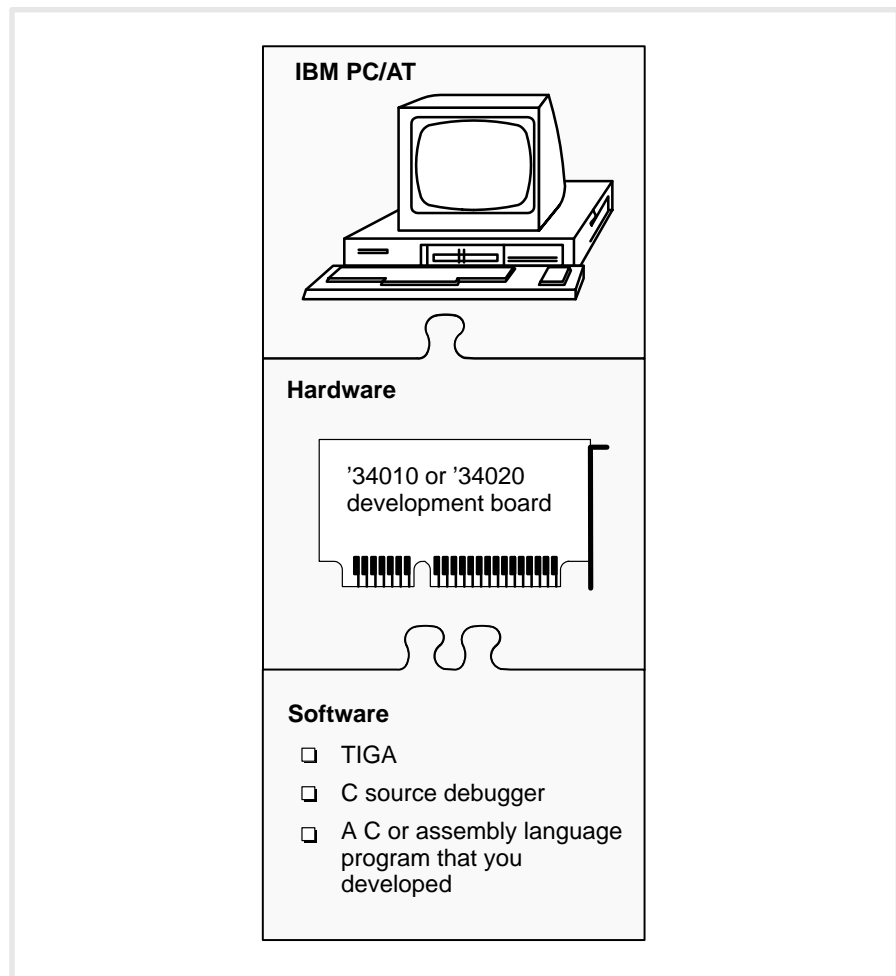
The development board version of the debugger can be used with the following boards:

❑ **'34010 TIGA development board (TDB).** This board is a standard video display adapter for ISA- and EISA-based PCs. The TDB provides an environment for debugging '34010 application software.

❑ **'34020 software development board (SDB).** This board is a standalone, high-performance, graphics-development board, compatible with the IBM PC-AT bus (ISA compatible). The SDB provides an environment for debugging '34020 application software.

❑ **Third-party '34010- or '34020-based boards** that use version 2.05 (or later) of the TIGA software interface.

Where are the installation instructions?
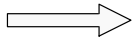
To install the development board version of the debugger, read Chapter 2, *Installing the Debugger for Use With '340 Development Boards.*

Chapter 2 assumes that you have already installed your board and that you have also installed TIGA:

❑ **Installing the '34010 TDB:** Refer to the *TMS34010 TIGA Development Board User's Guide*, which accompanies the TDB.

❑ **Installing the '34020 SDB:** Refer to the *TMS34020 Software Development Board User's Guide*, which accompanies the SDB.

❑ **Installing third-party boards:** Refer to the manufacturer's instructions accompanying your board.

❑ **Installing TIGA:** The documentation for installing TIGA usually accompanies your development board. If this is not available, refer to the *TIGA™ Interface User's Guide*.

1.2 Using the Emulator Version of the Debugger

Figure 1–2 shows the key hardware and software items that you'll need for using the emulator version of the C source debugger.

Figure 1–2. Key Items in an Emulator System
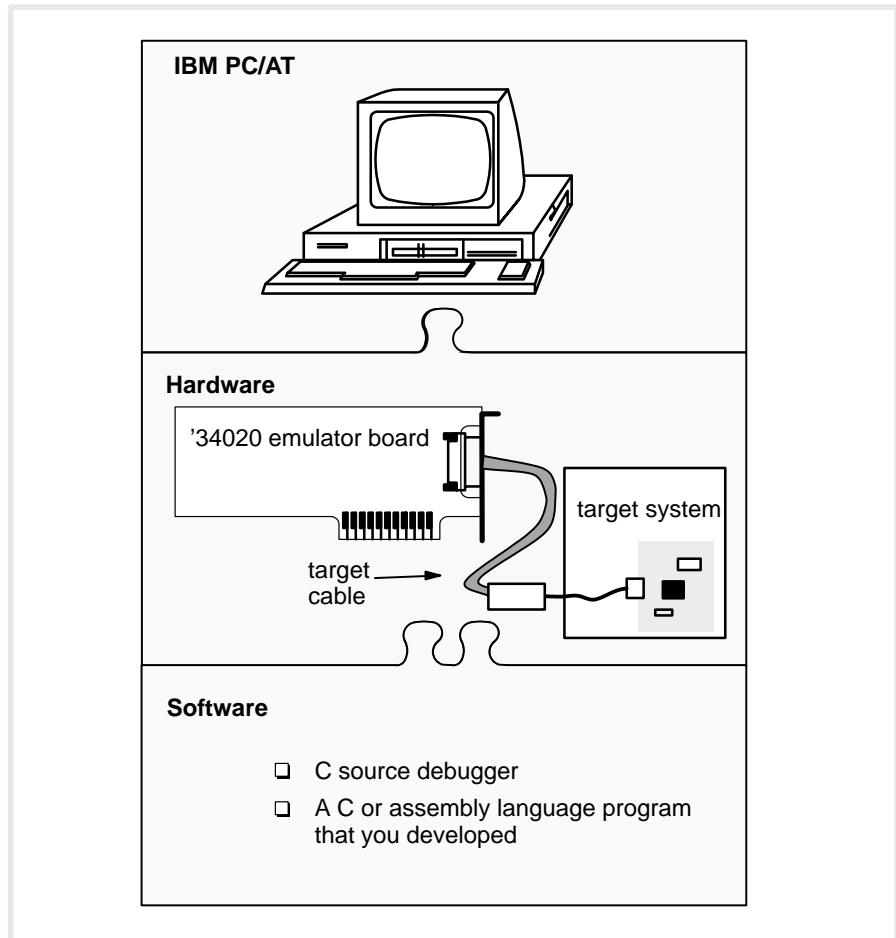
You can use this version of the debugger with the Texas Instruments '34020 emulator only. The emulator uses unique, scan-based TI emulation technology and provides an environment for debugging code on a '34020 board. The emulator does not contain a '34020 processor, so it must be connected to a target system that has a '34020. Usually, the target system is a board of your own design; for testing purposes, however, you can use the '34020 SDB as a target system.

Where are the installation instructions?
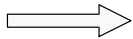
To install the emulator version of the debugger, read Chapter 3, *Installing the Debugger for Use With the '34020 Emulator.*

Chapter 3 assumes that you have already installed the emulator board and your target system. For emulator installation instructions, refer to the *TMS34020 Emulator Board Installation Guide,* which accompanies the emulator version of the debugger.

1.3 Strategies for Debugging Your '340 Application

'340 applications fall into three general categories:

❑ Standalone applications
❑ TIGA applications
❑ Non-TIGA host applications

The '340 C source debugger cannot be used with all three types of applications. The following paragraphs describe these application categories and tell you whether or not the debugger can be used.

Standalone applications

Standalone applications execute entirely on the '340 processor. The program you are debugging is composed of a single COFF object module. Only the debugger loads and executes your program; there is no interaction between your program and the PC.

Both versions of the C source debugger can be used without restriction in standalone environments:

❑ Because your program does not interact with the PC, the development board debugger is free to use the '340 processor's host interface (via the TIGA communication driver routines).

❑ The emulator version of the debugger does not use or need the '340 processor's host interface. The emulator's scan-based interface allows it to control the '340 system.

TIGA applications

For the purposes of debugging, a TIGA application is an application in which you have written your own TIGA modules to supplement the standard functions in the TIGA applications interface library. You can use the debugger to refine your custom modules. In a TIGA application, the debugging environment is split into two parts.

❑ The TIGA application resides on a target PC, with TIGA; TIGA relocates code and loads it into '340 memory.

❑ The debugger software resides on a host PC.

These two parts must communicate with each other during program execution. If you are using the emulator, the two parts communicate via the emulation cable. If you are using a development system, the two parts must communicate over a serial cable.

Both the emulator and development board versions of the debugger allow you to debug TIGA applications. For more information about debugging TIGA applications, refer to Chapter 10, *Loading TIGA Applications.*

Non-TIGA host applications

Non-TIGA host applications are similar to TIGA applications because they are split into two parts:

❏ One part, like a standalone application, executes entirely on the '340 processor.

❏ Another part executes on the PC.

These two parts must communicate with each other during program execution.

The emulator version of the C source debugger can be used with non-TIGA host applications as long as the debugger runs on a different PC or runs as a separate task under a multitasking DOS extension. The debugger controls the '340 processor through the emulator interface, leaving the host port free for controlling communications between the two parts of the application software. If the host system, rather than the emulator, loads the program that you plan to debug, the debugger must load the symbols for the '340 program (with the SLOAD debugger command or the debugger –s option).

The development board version of the C source debugger is not compatible with non-TIGA host applications, because the debugger requires exclusive use of the host interface.

1.4   Using an Emulator and a Development Board in the Same System

An emulator and a development board are not often used in the same system—usually, you will want to use one or the other for debugging your '340 application. However, it's possible that you may want to use both an emulator and a development board. For example, you may be using the '34020 SDB as a target system for the emulator.

If you use an emulator and a development board in the same system, it's important to note that:

❏ The debugger provides nearly identical functionality for the development boards and for the emulator. Remember, though, that there are two different versions of the debugger, and they are invoked with different commands:

db340   Is the command that invokes the development board version of the debugger.

db340emu   Is the command that invokes the emulator version of the debugger.

If you are using the SDB as a target system for the emulator, you should use the emulator version of the debugger.

❏ If you use a development board separately from the emulator, you must be sure that the emulator is running free while you are using the development board version of the debugger. To do this, invoke the emulator debugger, enter the RUNF command, then quit. Now you will be able to invoke the development board debugger without interference from the emulator.

If the emulator is not running free while you are using the development board debugger, the development board's '340 processor could be halted by the emulator. This prevents the development board software from gaining control of the '340.

Installing the Debugger for Use With '340-Based Development Boards

STOP If you are using the debugger with the '34020 emulator, do not follow the installation instructions in this chapter—turn to Chapter 3. If you are not sure which version of the debugger you should install, read Chapter 1.

This chapter will help you install the development board version of the '340 C source debugger. This version of the debugger works with '34010- and '34020-based PC boards that use TIGA (version 2.05 or later). For example, it works with the '34010 TDB and the '34020 SDB.

In most cases, if you install the debugger as instructed in this chapter, it will operate properly. However, if your debugger doesn't seem to work properly, or if you are developing a nonstandard or advanced application, refer to Appendix A, *Technical Notes*, for troubleshooting and supplementary information.

When you finish installing the debugger, turn to Chapter 4, *An Introductory Tutorial to the C Source Debugger.*

2.1   What You'll Need

In addition to the items shipped with the C source debugger, you'll need the following items.

Hardware checklist

☐	**host**	An IBM PC/AT or 100% compatible ISA/EISA-bus PC with a hard-disk system and a 1.2M floppy-disk drive
☐	**target**	If you plan to debug TIGA applications, you will also need a target PC to run TIGA and the TIGA application
☐	**memory**	Minimum of 640K (debugger occupies approximately 400K)
☐	**display**	Monochrome or color (color recommended)
☐	**development board**	A '340-based PC board
☐	**optional hardware**	Mouse (must be compatible with a Microsoft mouse)
☐		An EGA- or VGA-compatible graphics display card
☐		A 17" or 19" monitor. The C source debugger has several modes that allow you to display varying amounts of information on your PC monitor. If you have an EGA- or VGA-compatible graphics card and a large monitor (17" or 19"), you can take advantage of some of the debugger's larger screen modes. (To use larger screen sizes, you must invoke the debugger with the appropriate options; Table 2–1, page xii, explains this in detail.)
☐		A second monitor. Most development boards allow you to connect the board to both your PC's monitor and to a second display monitor. The debugger display is shown on your PC's display monitor. If you want to display any graphics routines drawn by your '340 code, you must connect the development board to a second display monitor.
☐	**miscellaneous materials**	A blank, formatted disk

Software checklist

☐	**operating system**	MS-DOS or PC-DOS (version 3.0 or later)
☐	**software tools**	'340 family C compiler, assembler, and linker (version 5.0 or later)

☐ **required files** † *gspmon.out* contains '340 routines that the debugger uses for controlling the '340 processor.

☐ † *tigacom.exe* and *debugcom.exe* are two communication drivers required for serial communication in TIGA applications.

☐ ‡ *tigacd* loads the TIGA communication driver.

☐ ‡ If you plan to debug TIGA applications, you'll need *tigagm.out* (the TIGA graphics manager).

☐ ‡ If you are debugging extended primitives, you also need *extprims.rlm.*

☐ **optional files** † *dbinit.cmd* is a general-purpose batch file that contains debugger commands. When you invoke the debugger, it will execute the commands in dbinit.cmd.

Initially, dbinit.cmd disables memory mapping; if dbinit.cmd isn't present when you invoke the debugger, then all memory is invalid at first. Two additional files, *sdbmap.cmd* and *tdbmap.cmd*, define sample memory maps. When you first start to use the debugger, it is usually not necessary to enable memory mapping. Later, you may want to define your own memory map. For information about these files and about setting up your own memory map, refer to Chapter 8, *Defining a Memory Map*.

☐ † *init.clr* is a general-purpose screen configuration file. If init.clr isn't present when you invoke the debugger, the debugger uses the default screen configuration.

The default configuration is for color monitors; an additional file, *mono.clr*, can be used for monochrome monitors. When you first start to use the debugger, the default screen configuration should be sufficient for your needs. Later, you may want to define your own custom configuration. For information about these files and about setting up your own screen configuration, refer to Chapter 13, *Customizing the Display*.

† Included as part of the debugger package
‡ Included as part of the TIGA software package

Note that the debugger operates correctly in a Windows 3.0 environment; however, the mouse may not function properly.

You **must** use the TIGA software interface (version 2.05 or later). The installation instructions in this chapter assume that you have already installed TIGA according to the instructions provided with your third-party development board or in the *TIGA™ Interface User's Guide.*

2.2 Installing the Debugger Software

This section explains the simple process of installing the debugger software on a hard disk system. The debugger package includes a single disk that contains multiple directories. To install the debugger, you must copy the *db* directory from the product disk.

Step 1: Make a backup copy of the product disk. (If necessary, refer to the DOS manual that came with your computer.)

Step 2: On your hard disk or system disk, create a directory named *db*. This directory will contain the '340 C source debugger software.

```
MD C:\db ⏎
```

Step 3: Insert the product disk into drive A. Copy the contents of the db directory:

```
COPY A:\db\*.* C:\db\*.* /V ⏎
```

2.3 Setting Up the Debugger Environment

To ensure that your debugger works correctly, you must perform several steps:

1) Modify the PATH statement to identify the db directory.
2) Define environment variables so that the debugger can find the files it needs.
3) Install the TIGA communication driver.

Not only must you do these things before you invoke the debugger for the first time, *you must do them any time you power up or reboot your PC.*

You can accomplish these tasks by entering individual DOS commands, but it's simpler to put the commands in a batch file. You can edit your system's autoexec.bat file; in some cases, modifying the autoexec may interfere with other applications running on your PC. So, if you prefer, you can create a separate batch file that performs these tasks.

Figure 2–1 *(a)* shows an example of an autoexec.bat file that contains the suggested modifications (highlighted in bold type). Figure 2–1 *(b)* shows a sample batch file that you could create instead of editing the autoexec.bat file. (For the purpose of discussion, assume that this sample file is named *initdb.bat.*)

Figure 2–1. DOS-Command Set Up for the Debugger

(a) Sample autoexec.bat file to use with the debugger

Modifications:

PATH statement ⟶

Environment variables ⟶

Install TIGA driver ⟶

```
DATE
TIME
ECHO OFF
PATH=C:\DOS;C:\340TOOLS;C:tiga;C:\db
SET D_DIR=C:\db
SET D_SRC=;C:\340code
SET D_OPTIONS=-b
SET C_DIR=C:\340TOOLS
CLS
tigacd
```

(b) Sample initdb.bat file to use with the debugger

PATH statement ⟶

Environment variables ⟶

Install TIGA driver ⟶

```
PATH=C:\db;%path%
SET D_DIR=C:\db
SET D_SRC=C:\340code
SET D_OPTIONS=-b
tigacd
```

Invoking the new or modified batch file

❏ If you modify the autoexec.bat file, be sure to invoke it before invoking the debugger for the first time. To invoke this file, enter:

autoexec ⏎

❏ If you create an initdb.bat file, you must invoke it before invoking the debugger for the first time. After that, you'll need to invoke initdb.bat any time that you power up or reboot your PC. To invoke this file, enter:

initdb ⏎

Modifying the PATH statement

Step 1: Define a path to the debugger directory. The general format for doing this is:

`PATH=C:\db`

This allows you to invoke the debugger without specifying the name of the directory that contains the debugger executable file.

❑ If you are modifying an autoexec that already contains a PATH statement, simply include `;C:\db` at the end of the statement as shown in Figure 2–1 *(a)*.

❑ If you are using the initdb.bat file, use a different format for the PATH statement:

`PATH=C:\db;%path%`

The addition of `;%path%` ensures that this PATH statement won't undo PATH statements in any other batch files (including the autoexec.bat file).

Setting up the environment variables

An environment variable is a special system symbol that the debugger uses for finding or obtaining certain types of information. The debugger uses three environment variables, named D_DIR, D_SRC, and D_OPTIONS. The next three steps tell you how to set up these environment variables. The format for doing this is the same for both the autoexec.bat file and initdb.bat files.

Step 2: Set up the D_DIR environment variable to identify the db directory:

`SET D_DIR=C:\db`

(Be careful not to precede the equal sign with a space.)

This directory contains auxiliary files (such as dbinit.cmd) that the debugger needs.

Step 3: Set up the D_SRC environment variable to identify any directories that contain program source files that you'll want to look at while you're debugging code. The general format for this is:

SET D_SRC=C:*pathname$_1$;pathname$_2$...*

(Be careful not to precede the equal sign with a space.)

For example, if your '340 programs were in a directory named *340code*, the D_SRC set up would be:

`SET D_SRC=C:\340code`

Step 4: You can use several options when you invoke the debugger. If you use the same options over and over, it's convenient to specify them with D_OPTIONS. The general format for doing this is:

SET D_OPTIONS= [*object filename*] [*debugger options*]

(Be careful not to precede the equal sign with a space.)

This tells the debugger to load the specified object file and use the specified options each time you invoke the debugger. These are the options that you can identify with D_OPTIONS:

–b[bbbb]	–i *pathname*
–s	–v

For more information about debugger options, see Section 2.5 (page xii). Note that you can override D_OPTIONS by invoking the debugger with the –x option.

Installing the TIGA communication driver

Step 5: Install the TIGA communication driver by entering:

TIGACD ⏎

Note: TIGA Applications

If you will be debugging a TIGA application, you may want to wait to install the TIGA communication driver until you have read the information in Section 2.4, *Setting Up Serial Communications for TIGA Applications.*

2.4 Setting Up for TIGA Applications

If you plan to debug your own custom TIGA modules, you will need to run your application and the debugger on separate PCs. *If you do not plan to debug your own custom TIGA modules, skip this section; turn to Section 2.5, Invoking the Debugger, on page xii.*

Figure 2–2 shows the various components that are needed on the TIGA (target) side and the debugger (host) side.

Figure 2–2. Components Required for Debugging a TIGA Application

Target PC	Host PC

❏ '340 development board
❏ tigacd driver
     (TIGA version 2.05)
❏ tigacom driver
❏ Object code and
     target RLMs
❏ tigagm.out
❏ extprims.rlm
	❏ '340 debugger
	❏ debugcom driver
	❏ gspmon.out
	❏ dbinit.cmd
	❏ gspsetup.exe
	❏ Object code and
	     target RLMs
	❏ Source files
	❏ tigagm.out
	❏ extprims.rlm

RS–232 serial cable
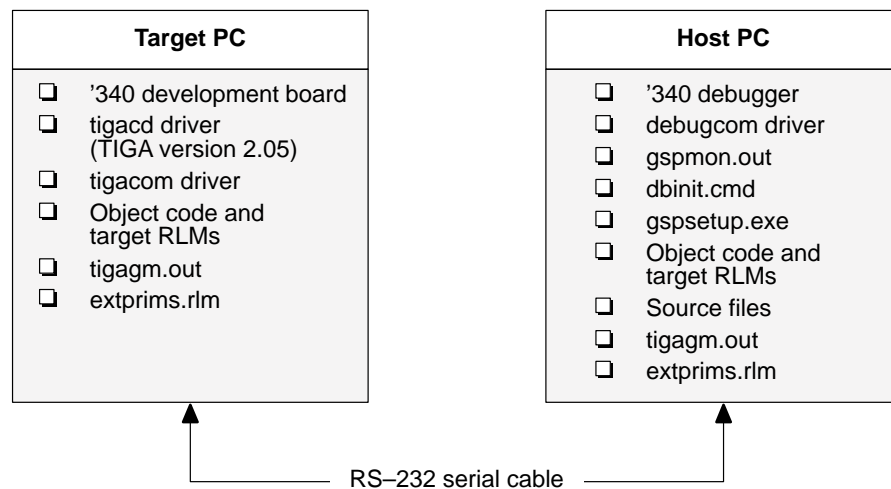
This system requires two serial communication drivers, tigacom and debugcom:

debugcom emulates portions of the TIGA communication driver (enabling the debugger to operate normally) and communicates with tigacom over the serial port.

tigacom receives commands from debugcom and relays them to TIGA. Any return values from TIGA are sent back across the serial link to debugcom, which, in turn, relays these values to the debugger.

Figure 2–3 (page xi) illustrates pinout connections for the host and target serial-cable connections.

Follow these steps to set up the serial link between your TIGA application and the debugger:

Step 1: Connect the target and host PCs with an RS–232 serial cable.

Step 2: Copy tigacom.exe from the product diskette to the target PC; put tigacom.exe in the directory where the TIGA communication driver resides.

Step 3: On the target PC, reinstall the TIGA communication driver:

`TIGACD /D2` ⏎

This enables the development-board version of the communication driver's debug mode. Later, if you need to disable this mode, enter:

`TIGACD /D0` ⏎

Step 4: On the target PC, install the tigacom driver by entering:

tigacom [**C**communication port] [**B**baud rate]

where communication port and baud rate are single digits. The communication port parameter can have any of these values:

Parameter value	Represents this baud rate
1	COM1 (3F8h)—**default**
2	COM2 (2F8h)
3	COM3 (3E8h)
4	COM4 (2E8h)

The baud rate parameter can have any of these settings:

Parameter value	Represents this baud rate	Parameter value	Represents this baud rate
1	1200	**5**	19200—**default**
2	2400	**6**	38400
3	4800	**7**	57600
4	9600	**8**	115200

Note: TIGA Communication Driver

If you ever uninstall the TIGA communication driver (by entering **`tigacd /u`**) and then reinstall it, you must also reinstall tigacom. The tigacom driver will uninstall itself whenever it detects that the TIGA communication driver has been unloaded.

Step 5: On the host PC, install the debugcom driver by entering:

debugcom   [**C***communication port*] [**B***baud rate*]

where *communication port* and *baud rate* can have the same values as those listed for debugcom. You can use different COM port settings for the two PCs, but the baud rates must be the same.

Step 6: For each module that you plan to debug, copy the source and object code to the host PC. The source code can go in any directory, but the object code *must* reside in the same directory as db340.exe (the db directory).

Step 7: Be sure that you have copied tigagm.out and, if you are using TIGA extended primitives, extprims.rlm to the directory on the target PC that contains tigacd. Also copy them to the db directory on the host PC.

Step 8: On the host PC, execute the gspsetup utility to verify that the serial link is operating properly. Enter:

`gspsetup -D` ⏎

This will print the status of several tests. If no errors are reported, you are ready to invoke the debugger. If gspsetup does not execute properly, refer to *Common serial link problems* on page A-4.

Step 9: On the host PC, invoke the debugger:

`db340` ⏎

The debugger should come up in a state where the '340 processor is in an endless loop. Use the RUN command (or press F5) to start the processor running.

Step 10: Verify that the debugger is working properly by reloading the TIGA graphics manager. On the target PC, enter:

`tigalnk /lx` ⏎

The debugger should display a message indicating that it has successfully loaded the TIGA graphics manager.

If "File Not Found" types of errors show up on the host PC, check your setup against the illustration in Figure 2–2 (page viii). Be sure that all of the files have been copied to the correct directories and that the serial cable is set up correctly.

If the tigalnk utility produces other types of errors, you will need to reboot the target PC and begin again with Step 4, using lower baud rate settings for tigacom and debugcom.

You may have to try several baud rates until you determine which is the highest baud rate you can use for reliable operation. In most

systems, the baud rate can be set as high as 115200 baud, but in some 386 protected mode systems, mode switching restrictions result in a maximum baud rate of 37400 to 57600 baud. The best method for determining the optimum baud rate is to start at 115200 baud and work down until serial communications are stable.

Like the information in Section 2.3, you have to install tigacd, tigacom, and debugcom whenever you power up or reboot the PC. Once you have determined the correct baud rates for tigacom and debugcom, you may want to enter the tigacd and tigacom commands into the target PC's autoexec and enter the debugcom command into the host PC's autoexec (or whatever initialization file you choose to use).

Pinout for serial cable connector

Figure 2–3 illustrates the pinout configuration for the RS-232 serial cable used for communications between the host and target PCs. Note that pinouts are shown for both a 9-pin and a 25-pin connector.,

Figure 2–3. Serial Cable Pinout Connections

(a) Connections for a 9-pin connector   *(a) Connections for a 25-pin connector*

Host PC Connector		Target PC Connector
pin 2	◄►	pin 3
pin 3	◄►	pin 2
pin 5	◄►	pin 5
		pins 7–8
		pins 1–4–6

Host P2 Connector		Target PC Connector
pin 3	◄►	pin 2
pin 2	◄►	pin 3
pin 7	◄►	pin 7
		pins 4–5
		pins 6–8–20
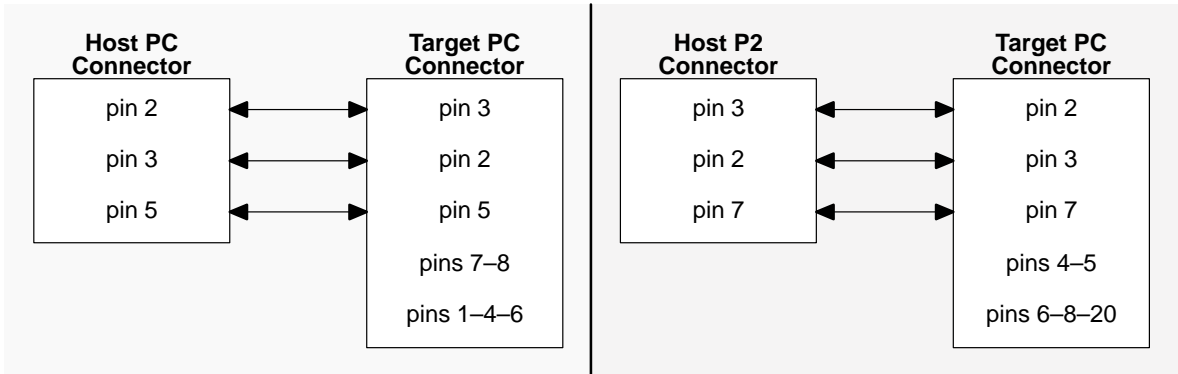
Notes: 1) **9-pin connector**: On the target end, pins 7 and 8 are tied together and pins 1, 4, and 6 are tied together.

2) **25-pin connector**: On the target end, pins 4 and 5 are tied together and pins 6, 8 and 20 are tied together.

2.5   Invoking the Debugger

Here's the basic format for the command that invokes the debugger:
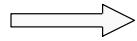
db340   [*filename*]   [*−options*]

db340   is the command that invokes the development board version of the debugger.

filename   is an optional parameter that names an object file that the debugger will load into memory during invocation. The debugger looks for the file in the current directory; if the file isn't in the current directory, you must supply the entire pathname. If you don't supply an extension for the filename, the debugger assumes that the extension is .out.

−options   supply the debugger with additional information (see Table 2−1).

You can also specify filename and option information with the D_OPTIONS environment variable (see *Setting up the environment variables*, page vi).

Table 2−1. Debugger Options

Option	Description
−b[bbbb]	**Screen-size options.** By default, the debugger uses an 80-character-by-25-line screen. If you have a special graphics card, however, you can choose one of several larger screen sizes.

Option	Characters/Lines	Notes
none	80 by 25	This is the default display
−b	80 by 43 (EGA)	Use any EGA or VGA card
	80 by 50 (VGA)	
−bb	120 by 43	
−bbb	132 by 43	Currently, the debugger supports these modes on a Video Seven VEGA Deluxe card.
−bbbb	80 by 60	
−bbbbb	100 by 60	

Option	Description
−i *pathname*	**Additional directories**. Replace *pathname* with an appropriate directory name. You can specify several pathnames; use the −i option as many times as necessary:

db340   **−i** *path$_1$*   **−i** *path$_2$*   **−i** *path$_3$* . . .

Using −i is similar to using the D_SRC environment variable (described on page vi). If you name directories with both −i and D_SRC, the debugger first searches through directories named with −i. The debugger can track a cumulative total of 20 paths (including paths specified with D_SRC and the debugger USE command).

Option	Description
−mc	**'34082 support.** −mc tells the debugger to provide '34082 support. This allows you to access '34082 register values and tells the debugger to disassemble '34082 coprocessor instructions.

Table 2–1. Debugger Options (Continued)

Option	Description
–mi	**Don't initialize PC or SP.** By default, the debugger automatically initializes the '340 processor's program counter (PC) and stack pointer (SP—register A15/B15) to a section of memory assigned to these registers by the TIGA memory manager. The –mi option allows you to leave the PC and SP uninitialized. However, if the PC and SP are not pointing to valid RAM, you will not be able to invoke the debugger.
–mf	**Floating-point format.** By default, the debugger expects source code to use '340 floating-point format. The –mf option tells the debugger to expect IEEE floating-point format (IEEE std 754-1985) instead.
	If you are already using –mc, you don't need to use –mf.
–s	**Load symbol table only.** If you supply a *filename* when you invoke the debugger, you can use the –s option to tell the debugger to load only the file's symbol table (without the file's object code). This is similar to the debugger's SLOAD command.
–v	**Load without symbol table.** This option prevents the debugger from loading the entire symbol table when you load an object file. The debugger loads only the global symbols and later loads local symbols as it needs them. This speeds up the loading time and consumes less memory space.
	The –v option affects all loads, including loading when you invoke the debugger and loading with the LOAD command within the debugger environment.
–x	**Ignore D_OPTIONS.** –x tells the debugger to ignore any information supplied with D_OPTIONS.
–t *filename*	**New initialization file.** The –t option allows you to specify an initialization command file that will be used instead of dbinit.cmd.

2.6 Exiting the Debugger

To exit the debugger and return to the operating system, enter this command:

quit ⏎

You don't need to worry about where the cursor is or which window is active—just type. If a program is running, press (ESC) to halt program execution before you quit the debugger.

Installing the Debugger
for Use With the '34020 Emulator

STOP

If you aren't using the debugger with the '34020 emulator, do not follow the installation instructions in this chapter—turn to Chapter 2. If you are not sure which version of the debugger you should install, read Chapter 1.

This chapter will help you install the emulator version of the '340 C source debugger. This version of the debugger works only with the '34020 emulator.

In most cases, if you install the debugger as instructed in this chapter, it will operate properly. However, if your debugger doesn't seem to work properly, refer to Appendix A, *Technical Notes.* Also, if you are developing a nonstandard or advanced application, refer to Appendix A for supplementary information.

When you finish installing the debugger, turn to Chapter 4, *An Introductory Tutorial to the C Source Debugger.*

3.1 What You'll Need

In addition to the items that are shipped with the '340 C source debugger, you'll need the following.

Hardware checklist

☐ **host** An IBM PC/AT or 100% compatible ISA/EISA-bus PC with a hard-disk system and a 1.2M floppy-disk drive

☐ **target** If you plan to debug TIGA applications, you will also need a target PC to run TIGA and the TIGA application

☐ **memory** Minimum of 640K (debugger occupies approximately 400K)

☐ **display** Monochrome or color (color recommended)

☐ **emulator system** The '34020 emulator board and a '34020 target system

☐ **optional hardware** Mouse (must be compatible with a Microsoft mouse)

☐ An EGA- or VGA-compatible graphics display card

☐ A 17" or 19" monitor. The C source debugger has several modes that allow you to display varying amounts of information on your PC monitor. If you have an EGA- or VGA-compatible graphics card and a large monitor (17" or 19"), you can take advantage of some of the debugger's larger screen modes. (To use larger screen sizes, you must invoke the debugger with the appropriate options; Table 3–3, page xii, explains this in detail.)

☐ **miscellaneous materials** A blank, formatted disk

Software checklist

☐	**operating system**	MS-DOS or PC-DOS (version 3.0 or later)
☐	**software tools**	'340 family C compiler, assembler, and linker (version 5.0 or later)
☐	**required files** †	*emurst* resets the '34020 emulator
☐	‡	If you plan to debug TIGA applications, you'll need *tigagm.out* (the TIGA graphics manager).
☐	‡	If you are debugging extended primitives, you also need *extprims.rlm.*
☐	**optional files** †	*emuinit.cmd* is a file that a contains debugger commands.When you invoke the debugger, it will execute the commands in emuinit.cmd. Initially, emuinit.cmd disables memory mapping; if emuinit.cmd isn't present when you invoke the debugger, then all memory is invalid at first. When you first start to use the debugger, it is usually not necessary to enable memory mapping. Later, you may want to define your own memory map. For information about setting up your own memory map, refer to Chapter 8, *Defining a Memory Map*.
☐	†	*init.clr* is a general-purpose screen configuration file. If this file isn't present when you invoke the debugger, the debugger uses the default screen configuration.

init.clr additional para: The default configuration is for color monitors; an additional file, *mono.clr*, can be used for monochrome monitors.When you first start to use the debugger, the default screen configuration should be sufficient for your needs. Later, you may want to define your own custom configuration. For information about these files and about setting up your own screen configuration, refer to Chapter 13, *Customizing the Display*.

† Included as part of the debugger package
‡ Included as part of the TIGA software package

The debugger operates correctly in a Windows 3.0 environment; however, the mouse may not function properly.

3.2 Installing the Debugger Software

This section explains the simple process of installing the debugger software on a hard disk system. The debugger package includes a single disk that contains multiple directories. To install the debugger, you must copy the *emu34020* directory from the product disk.

Step 1: Make a backup copy of the product disk. (If necessary, refer to the DOS manual that came with your computer.)

Step 2: On your hard disk or system disk, create a directory named *emu34020*. This directory will contain the '34020 debugger software.

```
MD C:\emu34020  ⏎
```

Step 3: Insert the product disk into drive A. Copy the contents of the emu34020 directory:

```
COPY A:\emu34020\*.* C:\emu34020\*.* /V  ⏎
```

3.3   Setting Up the Debugger Environment

To ensure that your debugger works correctly, you must:

1)   Modify the PATH statement to identify the emu34020 directory.
2)   Define environment variables so that the debugger can find the files it needs.
3)   Identify any nondefault I/O space used by the emulator.
4)   Reset the emulator.
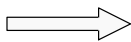
Not only must you do these things before you invoke the debugger for the first time, *you must do them any time you power up or reboot your PC.*

You can accomplish these tasks by entering individual DOS commands, but it's simpler to put the commands in a batch file. You can edit your system's autoexec.bat file; in some cases, however, modifying the autoexec may interfere with other applications running on your PC. So, if you prefer, you can create a separate batch file that performs these tasks.

Figure 3–1 *(a)* shows an example of an autoexec.bat file that contains the suggested modifications (highlighted in bold type). Figure 3–1 *(b)* shows a sample batch file that you could create instead of editing the autoexec.bat file. (For the purpose of discussion, assume that this sample file is named *initdb.bat*.) The subsections following the figure describe these commands.

Figure 3–1. DOS-Command Set Up for the Debugger

(a) Sample autoexec.bat file to use with the debugger

Modifications:	DATE
	TIME
	ECHO OFF
PATH statement →	**PATH=**C:\DOS;C:\340TOOLS;**C:\emu34020**
Environment variables →	**SET D_DIR=C:\emu34020;C:\cmdfiles**
	SET D_SRC=C:\340source
Identify I/O space →	**SET D_OPTIONS= –b**
	SET C_DIR=C:\340tools
Reset the emulator →	CLS
	emurst

(b) Sample initdb.bat file to use with the debugger

PATH statement →	PATH=C:\emu34020;%path%
Environment variables →	SET D_DIR=C:\emu34020
	SET D_SRC=C:\340source
Identify I/O space →	SET D_OPTIONS= –b
Reset the emulator →	emurst

Invoking the new or modified batch file

❑ If you modify the autoexec.bat file, you must invoke the file before invoking the debugger for the first time. To invoke this file, enter:

autoexec ⏎

❑ If you create an initdb.bat file, you must invoke the file before invoking the debugger for the first time. After that, you must invoke initdb.bat any time that you power up or reboot your PC. To invoke this file, enter:

initdb ⏎

Modifying the PATH statement

Step 1: Define a path to the debugger directory. The general format for doing this is:

`PATH=C:\emu34020`

This allows you to invoke the debugger without specifying the name of the directory that contains the debugger executable file.

❑ If you are modifying an autoexec that already contains a PATH statement, simply include `;C:\emu34020` at the end of the statement as shown in Figure 3–1 *(a)*.

❑ If you are using the initdb.bat file, use a different format for the PATH statement:

`PATH=C:\emu34020;%path%`

The addition of `;%path%` ensures that this PATH statement won't undo PATH statements in any other batch files (including the autoexec.bat file).

Setting up the environment variables

An environment variable is a special system symbol that the debugger uses for finding or obtaining certain types of information. The debugger uses three environment variables, named D_DIR, D_SRC, and D_OPTIONS. The next three steps tell you how to set up these environment variables. The format for doing this is the same for both the autoexec.bat and initdb.bat files.

Step 2: Set up the D_DIR environment variable to identify the emu34020 directory:

`SET D_DIR=C:\emu34020`

(Be careful not to precede the equal sign with a space.)

This directory contains auxiliary files (such as emuinit.cmd) that the debugger needs.

Step 3: Set up the D_SRC environment variable to identify any directories that contain program source files that you'll want to look at while you're debugging code. The general format for this is:

SET D_SRC=C:*pathname$_1$;pathname$_2$...*

(Be careful not to precede the equal sign with a space.)

For example, if you keep all of your C source files in a directory named CSOURCE, the line will look like this:

`SET D_SRC=C:\CSOURCE`

Step 4: You can use several options when you invoke the debugger. If you use the same options over and over, it's convenient to specify them with D_OPTIONS. The general format for doing this is:

SET D_OPTIONS= [*object filename*] [*debugger options*]

(Be careful not to precede the equal sign with a space.)

This tells the debugger to load the specified object file and use the specified options each time you invoke the debugger. These are the options that you can identify with D_OPTIONS:

–b[bbbb]	–p *port address*	–i *pathname*
–s		–v

For more information about debugger options, see Section 3.5 (page xii). Note that you can override D_OPTIONS by invoking the debugger with the –x option.

If you specify a –p option, it will be used by both the debugger and the emurst utility.

Identifying the correct I/O switches

Step 5: If you didn't modify the emulator's I/O switches when you installed the emulator board, skip this step.

If you modified the I/O switch settings, you must use the debugger's –p option to identify the I/O space that the emulator is using. You can do this each time you invoke the debugger, or you can specify this information by using the D_OPTIONS environment variable as shown in Table 3–1.

Table 3–1. Using D_OPTIONS to Identify Nondefault I/O Address Space

	switch #		Add this line to the
	1	**2**	**autoexec.bat or initdb.bat file**
0x0220–0x023F	on	on	SET D_OPTIONS=-P 220
0x0300–0x031F	off	on	SET D_OPTIONS=-P 300
0x03E0–0x03FF	off	off	SET D_OPTIONS=-P 3E0

Resetting the emulator

Step 6: To reset the emulator, invoke the emurst utility that comes with the debugger package. The general format for doing this is:

emurst [**–x**] [**–p** *port address*]

You can include the emurst command in the autoexec.bat or initdb.bat file; the command format is the same for either type of file.

If you didn't modify the I/O switches, it is not necessary to use either of emurst's parameters. If you modified the I/O switches, you can use the –p option as shown in Table 3–2. If you already specified –p with the D_OPTIONS environment variable, it is not necessary to use –p with the emurst utility.

Table 3–2. –x Options for the emurst Utility

	switch #		
	1	**2**	**Invoke emurst with this option**
0x0220–0x023F	on	on	`emurst -p 220`
0x0300–0x031F	off	on	`emurst -p 300`
0x03E0–0x03FF	off	off	`emurst -p 3E0`

If you set emurst's –p option with D_OPTIONS and need to override it, invoke emurst from the DOS prompt by using both –x and –p:

emurst -x -p *new port address*

This tells emurst to ignore the *port address* supplied by D_OPTIONS and to use the new one that was specified on the command line. A port address specified with emurst **must not** differ from the D_OPTIONS *port address* **unless** you also use emurst's –x option.

Note that running emurst places the emulator board and the target cable buffer pod in a disconnected state: the '34020 does not perform emulation functions and executes as if the emulator is not installed. Every time the target's power is cycled, the buffer pod automatically switches to the disconnect state. However, if the debugger is not running when target power is cycled, a mismatch state can occur between the pod and the controller. To avoid this problem, *always execute emurst after powering up the target system*. Also, after powering up the target system, you must reset the '34020 before you can use the debugger. If the debugger is running and you need to cycle the target power, enter a RUNF command first. These situations also apply to

applications where the target system and the emulator board are in different PCs.

3.4   Setting Up for TIGA Applications

If you plan to debug your own custom TIGA modules, you will need to run your application and the debugger on separate PCs. *If you do not plan to debug your own custom TIGA modules, skip this section; turn to Section 3.5, Invoking the Debugger, on page xii.*

Figure 3–2 shows the various components that are needed on the TIGA (target) side and the debugger (host) side.

Figure 3–2. Components Required for Debugging a TIGA Application

Target PC	Host PC

❏   '340 board
❏   tigacd driver
       (TIGA version 2.05)
❏   Object code and
       target RLMs
❏   tigagm.out
❏   extprims.rlm
	❏ '340 debugger
	❏ init.cmd
	❏ Object code and
	       target RLMs
	❏ Source files
	❏ tigagm.out
	❏ extprims.rlm

emulator serial cable
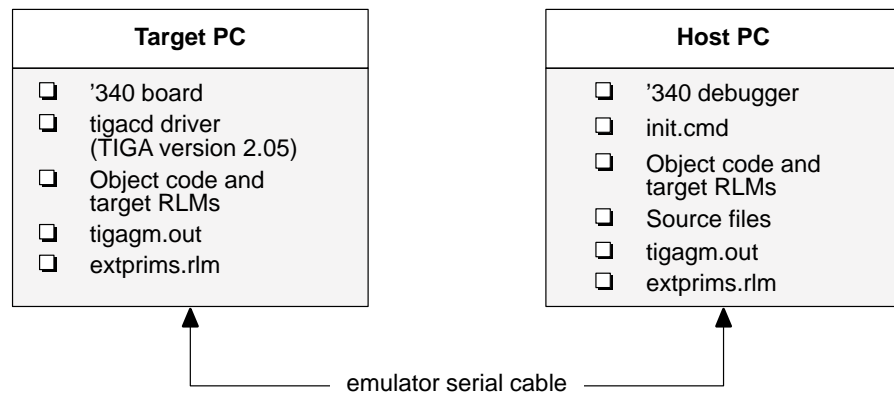
This system requires no special communication drivers. However, there are a few additional installation steps that you must follow:

Step 1:   Install the TIGA communications driver (tigacd) on the target PC. On the target PC, enter:

TIGACD /D1 ⏎

This enables the emulator version of the communication driver's debug mode. Later, if you need to disable this mode, enter:

TIGACD /D0 ⏎

Step 2:   For each module that you plan to debug, copy the source and object code to the host PC. The source code can go in any directory, but the object code *must* reside in the same directory as db340emu.exe (the emu34020 directory).

Step 3: Be sure that you have copied tigagm.out and, if you are using TIGA extended primitives, extprims.rlm to the directory on the target PC that contains tigacd. Also copy them to the emu34020 directory on the host PC.

Step 4: On the host PC, invoke the debugger:

`db340emu` ⏎

The debugger should come up in a state where the '340 processor is in an endless loop. Use the RUN command (or press F5) to start the processor running.

Step 5: Verify that the debugger is working properly by reloading the TIGA graphics manager. On the target PC, enter:

`tigalnk /lx` ⏎

The debugger should display a message indicating that it has successfully loaded the TIGA graphics manager.

If "File Not Found" types of errors show up on the host PC, check your setup against the illustration in Figure 3–2 (page x). Be sure that all of the files have been copied to the correct directories and that the serial cable is set up correctly.

3.5   Invoking the Debugger

Here's the basic format for the command that invokes the debugger:

⟹

> **db340emu**   [*filename*]   [*–options*]

db340emu   is the command that invokes the emulator version of the debugger.

filename   is an optional parameter that names an object file that the debugger will load into memory during invocation. The debugger looks for the file in the current directory; if the file isn't in the current directory, you must supply the entire pathname. If you don't supply an extension for the filename, the debugger assumes that the extension is .out.

–options   supply the debugger with additional information (see Table 3–3).

You can also specify filename and option information with the D_OPTIONS environment variable (see *Setting up the environment variables*, page vi).

Table 3–3. Debugger Options

Option	Description
–b[bbbb]	**Screen-size options.** By default, the debugger uses an 80-character-by-25-line screen. If you have a special graphics card, however, you can choose one of several larger screen sizes.
	<table><tr><td>Option</td><td>Characters/Lines</td><td>Notes</td></tr><tr><td>*none*</td><td>80 by 25</td><td>This is the default display</td></tr><tr><td>–b</td><td>80 by 43 (EGA)</td><td>Use any EGA or VGA card</td></tr><tr><td></td><td>80 by 50 (VGA)</td><td rowspan="5">Currently, the debugger supports these modes on a Video Seven VEGA Deluxe card.</td></tr><tr><td>–bb</td><td>120 by 43</td></tr><tr><td>–bbb</td><td>132 by 43</td></tr><tr><td>–bbbb</td><td>80 by 60</td></tr><tr><td>–bbbbb</td><td>100 by 60</td></tr></table>
–p *port address*	**Port address.** –p identifies the I/O port address that the debugger uses for communicating with the emulator. If you used the emulator's default switch settings, you don't need to use the –p option. **If you used nondefault switch settings, you must use –p**. Depending on your switch settings, replace *port address* with one of these values:
	<table><tr><td>Switch 1</td><td>Switch 2</td><td>Option</td></tr><tr><td>on</td><td>on</td><td>–p 220</td></tr><tr><td>on</td><td>off</td><td>*none needed* (default setting)</td></tr><tr><td>off</td><td>on</td><td>–p 300</td></tr><tr><td>off</td><td>off</td><td>–p 3E0</td></tr></table>
–s	**Load symbol table only.** If you supply a *filename* when you invoke the debugger, you can use the –s option to tell the debugger to load only the file's symbol table (without the file's object code). This is similar to the debugger's SLOAD command.

Table 3–3. Debugger Options (Continued)

Option	Description
–i *pathname*	**Additional directories.** –i identifies additional directories that contain your source files. Replace *pathname* with an appropriate directory name. You can specify several pathnames; use the –i option as many times as necessary: **db340emu –i** *path$_1$* **–i** *path$_2$* **–i** *path$_3$* . . . Using –i is similar to using the D_SRC environment variable (described on page vi). If you name directories with both –i and D_SRC, the debugger first searches through directories named with –i. The debugger can track a cumulative total of 20 paths (including paths specified with D_SRC and the debugger USE command).
–mc	**'34082 support.** –mc tells the debugger to provide '34082 support. This allows you to access '34082 register values and tells the debugger to disassemble '34082 coprocessor instructions.
–mf	**Floating-point format.** By default, the debugger expects source code to use '340 floating-point format. The –mf option tells the debugger to expect IEEE floating-point format (IEEE std 754-1985) instead. If you are already using –mc, it is not necessary to use –mf.
–v	**Load without symbol table.** This option prevents the debugger from loading the entire symbol table when you load an object file. The debugger loads only the global symbols and later loads local symbols as it needs them. This speeds up the loading time and consumes less memory space. The –v option affects all loads, including loading when you invoke the debugger and loading with the LOAD command within the debugger environment.
–x	**Ignore D_OPTIONS.** –x tells the debugger to ignore any information supplied with D_OPTIONS.
–t *filename*	**New initialization file.** The –t option allows you to specify an initialization command file that will be used instead of emuinit.cmd.

3.6 Exiting the Debugger

To exit the debugger and return to the operating system, enter this command:

quit ⏎

You don't need to worry about where the cursor is or which window is active—just type. If a program is running, press (ESC) to halt program execution before you quit the debugger.

An Introductory Tutorial
to the C Source Debugger

This chapter provides a step-by-step, hands-on demonstration of the '340 C source debugger's basic features. This is not the kind of tutorial that you can take home to read—this tutorial is effective only if you're sitting at your PC, performing the lessons in the order they're presented. This tutorial contains two sets of lessons (11 in the first, 14 in the second) and takes about one hour to complete.

Synopsis Page	Topic

How to use this tutorial

This tutorial contains three basic types of information:

Primary actions   Primary actions identify the main lessons in the tutorial; they're boxed so you can find them easily. A primary action looks like this:

> Make the CPU window the active window:
>
> **win CPU** [⏎]

Important information

In addition to primary actions, important information ensures that the tutorial works correctly. Important information is marked like this:

> **Important!** The CPU window should still be active from the previous step.

Alternative actions

Alternative actions show additional methods for performing the primary actions. Alternative actions are marked like this:

> **Try This:** Another way to display the current code in MEMORY is to show memory beginning from the current PC. . .

As you go through this tutorial, perform the primary actions and pay close attention to the important information. To learn even more about using the debugger, perform the alternative actions, too.

Important! This tutorial assumes that you have correctly and completely installed your development board or emulator (including invoking any files or DOS commands as instructed in the installation chapters).

A note about entering commands

Whenever this tutorial tells you to type a debugger command, just type—the debugger automatically places the text on the command line. You don't have to worry about moving the cursor to the command line; the debugger takes care of this for you. (There are a few instances when this isn't true—for example, when you're editing data in the CPU or MEMORY window—but this is explained later in the tutorial.)

Also, you don't have to worry about typing commands in uppercase or lowercase—either is fine. There are a few instances when a command's *parameters* must be entered in uppercase, and the tutorial points this out.

An escape route (just in case)

The steps in this tutorial create a path for you to follow. The tutorial won't purposely lead you off the path. But sometimes when people use new products, they accidently press the wrong key, push the wrong mouse button, or mistype a command. Suddenly, they're off the path without any idea of where they are or how they got there.

This probably won't happen to you. But, if it does, you can almost always get back to familiar ground by pressing (ESC). If you were running a program when you pressed (ESC), you should also type RESTART ⏎. Then go back to the beginning of whatever lesson you were in and try again.

Invoke the debugger and load the sample program's object code

Included with the debugger is a demonstration program named *sample*. This lesson shows you how to invoke the debugger and load the sample program. You will use the –b option so that the debugger uses a larger display.

Important! If you are using the emulator, this step assumes that you are using the default I/O switches or that you have identified the I/O switches with the D_OPTIONS environment variable (as described in the installation instructions in Chapter 3).

Invoke the debugger and load the sample program:

❏ For a **development board**, enter:

```
db340 –b c:\db\sample
```

❏ For the **emulator**, enter:

```
db340emu –b c:\emu34020\sample
```

Now what should I see?

Now you should see a display similar to this (it may not be exactly the same display, but it should be close).

menu bar with pulldown menus →

current PC (highlighted) →

reverse assembly of memory contents →

register contents →

```
Load     Brea     Watch    Memory   Color    MoDe     Run=F5    Step=F8    Next=F10
 DISASSEMBLY                                                      CPU
ffc00850 0740   c_int00:   SETF    32,0,0                         PC    ffc00850
ffc00860 09e9              MOVI    -3972128,A9                    ST    00000010
ffc00890 4d2d              MOVE    A9,A13                         A0    00000000
ffc008a0 4d2e              MOVE    A9,A14                         A1    00000000
ffc008b0 09e9              MOVI    -3908192,A9                    A2    00000000
ffc008e0 4d2f              MOVE    A9,SP                          A3    c0000010
ffc008f0 09ea              MOVI    -1,A10                         A4    00000000
ffc00920 0b4a              CMPI    -1,A10                         A5    00000000
ffc00940 ca05              JRZ     0ffc009a0h                     A6    00000000
ffc00950 a3cf              MOVE    A14,*-SP,1                     A7    00000040
ffc00960 934e              MOVE    A10,*A14+,1                    A8    00000000
ffc00970 0d5f              CALLA   var_init                       A9    ffc45da0
ffc009a0 0d5f              CALLA   main                           A10   ffffffff
ffc009d0 a3cf              MOVE    A14,*-SP,1                     A11   00001795
ffc009e0 182b              MOVK    1,A11                          A12   00000000
                                                                 A13   ffc363e0
                                                                 A14   ffc363e0
                                                                 A15   ffffa000
```

COMMAND window display area →

memory contents →

command line →

```
 COMMAND                                MEMORY
TMS340 Debugger Version 5.00    00000000 4000  4001  4002  4003  40004 4005  4006
Copyright (c) 1990, Texas In-   00000070 4007  4008  4009  400a  400b  400c  400d
strume                          000000e0 400e  400f  4010  4011  4012  4013  4014
TMS340x0 Development Board       00000150 4015  4016  4017  4018  4019  401a  401b
                                000001c0 401c  401d  401e  401f  4020  4021  4022
_                               00000230 4023  4024  4025  4026  4027  4028  4029
>>>
```

❑ If you **don't** see a display, then your development board or emulator may not be installed properly. Go back through the board installation instructions and be sure that you followed each step correctly; then reinvoke the debugger. If you still don't see a display, use the troubleshooting information in Appendix A.

❑ If you **do** see a display, *check the first few lines of the DISASSEMBLY window*. If these lines aren't the same—if, for example, they show ADD instructions—then enter the following commands on the debugger command line. (Just type; you don't have to worry about where the cursor is.)

1) Reset the '340 processor:

   reset ⏎

2) Load the sample program again:

   Development board: **load c:\db\sample** ⏎
   Emulator: **load c:\emu34020\sample** ⏎

What's in the DISASSEMBLY window?

The DISASSEMBLY window always shows the reverse assembly of memory contents; in this case, it shows an assembly language version of sample.out. The MEMORY window displays the current contents of memory. Because you loaded the object file sample.out when you invoked the debugger, memory contains the object code version of the sample file.

This tutorial step demonstrates that the code shown in the DISASSEMBLY window corresponds to memory contents. Initially, memory is displayed starting at address 0; if you look at the first line of the DISASSEMBLY window, you'll see that its display starts at address 0xFFC0 0850.

Modify the MEMORY display to show the same object code that is displayed in the DISASSEMBLY window:

`mem 0xffc00850` ⏎

Notice that the first column in the DISASSEMBLY window corresponds to the addresses in the MEMORY window; the second column in the DISASSEMBLY window corresponds to the memory contents displayed in the MEMORY window.

Try This: Another way to display the current code in MEMORY is to show memory beginning from the current PC:

`mem pc` ⏎

Select the active window

This lesson shows you how to make a window into the *active window.* You can move and resize any window; you can close some windows. Whenever you type a command or press a function key to move, resize, or close a window, the debugger must have some method of understanding which window you want to affect. The debugger does this by designating one window at a time to be the *active window.* Any window can be the active window, but only one window can be active at a time.

Make the CPU window the active window:

win CPU [⬚]

Important! If this didn't work, look at the way you entered the command. Did you enter **CPU** in uppercase letters? For this command, it's important that you enter the parameter in uppercase as shown.

Important! Notice the appearance of the CPU window (especially its borders) in contrast to the other, inactive windows. This is how you can tell which window is active.

Try This: Pressing the ⌜F6⌝ key "hops" through the windows in the display, making each one active in turn. Press ⌜F6⌝ as many times as necessary until the CPU window becomes the active window again.

Try This: You can also use the mouse to make a window active:

1) Point to any location on the window's border.

2) Click the left mouse button.

Be careful! If you point *inside* the window, the window becomes active when you press the mouse button, but something else may happen as well:

❏ If you're pointing inside the CPU window, then the register you're pointing at becomes active. The debugger then treats any text you type as a new value for that register. Similarly, if you're pointing inside the MEMORY window, the address you're pointing at becomes active.

Press ⌜ESC⌝ *to get out of this.*

❏ If you're pointing inside the DISASSEMBLY or FILE window, you'll set a breakpoint on the statement that you were pointing to.

Point to the same statement; press the button again to delete the breakpoint.

Resize the active window

This lesson shows you how to resize the active window.

Important! The CPU window should still be active from the previous step.

Make the CPU window as small as possible:

size 4,3 ⟳

This tells the debugger to make the window 4 characters by 3 lines, which is the smallest a window can be. (If it were any smaller, the debugger wouldn't be able to display all four corners of the window.) If you try to enter smaller values, the debugger will warn you that you've entered an *Invalid window size*. The maximum width and length depend on which –b option you used when you invoked the debugger. (If you'd like a complete list of the limits, see Table 6–1 on page 6-22.)

Make the CPU window larger:

size ⟳	*Enter the SIZE command without parameters*
↓ ↓ ↓	*Make the window 3 lines longer*
→ → → →	*Make the window 4 characters wider*
ESC	*Press this key when you finish sizing the window*

You can also use ↑ to make the window shorter and ← to make the window narrower.

Try This: You can also use the mouse to resize the window (note that this automatically causes the selected window to become the active window).

↖ 1) If you examine any window, you'll see a highlighted, backwards "L" in the lower right corner. Point to the lower right corner of the CPU window.

2) Press the left mouse button but don't release it; move the mouse while you're holding in the button. This resizes the window.

3) Release the mouse button when the window reaches the desired size.

Zoom the active window

Another way to resize the active window is to zoom it. Zooming the window makes it as large as possible.

Important! The CPU window should still be active from the previous steps.

Make the active window as large as possible:

zoom 📷

The window should now be as large as possible, taking up the entire display (except for the menu bar) and hiding all the other windows. You can "unzoom" the window by entering the ZOOM command again.

Return the window to its previous size. (Even though the COMMAND window is hidden by the CPU window, the ZOOM command will be recognized.)

zoom 📷

The window should now be back to the size it was before zooming.

Try This: You can also use the mouse to zoom the window.

Zoom the active window:

🮰   1)   Point to the upper left corner of the active window.

▌◻   2)   Click the left mouse button.

Return the window to its previous size by repeating these steps.

Move the active window

This lesson shows you how to move the active window.

Important! The CPU window should still be active from the previous steps.

Move the CPU window to the upper left portion of the screen:

move 0,1 🖉 *The debugger doesn't let you move the window to the very top—that would hide the menu bar*

The MOVE command's first parameter identifies the window's new X position on the screen. The second parameter identifies the window's new Y position on the screen. The maximum X and Y positions depend on which –b option you used when you invoked the debugger. (For a complete list of the limits, see Table 6–2 on page 6-25.)

Try This: You can use the MOVE command with no parameters and then use arrow keys to move the window:

move 🖉
→→→→ *Press → until the CPU window is back where it was (it may seem like only the border is moving—this is normal)*
ESC *Press ESC when you finish moving the window*

You can also use ⬆ to move the window up, ⬇ to move the window down, and ⬅ to move the window left.

Try This: You can also use the mouse to move the window (note that this automatically causes the selected window to become the active window).

↖ 1) Point to the top edge or left edge of the window border.

▐▌ 2) Press the left mouse button but don't release the button; move the mouse while you're holding in the button.

▐▌ 3) Release the mouse button when the window reaches the desired position.

Scroll through a window's contents

Many of the windows contain more information than can possibly be displayed at one time. You can view hidden information by moving through a window's contents. The easiest way to do this is to use the mouse to scroll the display up or down.

If you examine most windows, you'll see an up arrow near the top of the right border and a down arrow near the bottom of the right border. These are *scroll arrows*.

Scroll through the contents of the DISASSEMBLY window:

1) Point to either of the scroll arrows.

2) Press the left mouse button; continue pressing it until the display has scrolled several lines.

3) Release the button.

Try This: You can also use several of the keys to modify the display in the active window.

Make the MEMORY window the active window:

win MEMORY ⏎

Now try pressing these keys; observe their effects on the window's contents.

⬇   ⬆   PAGE DOWN   PAGE UP

These keys don't work the same for all windows; Section 14.3 (page 14-38) summarizes the functions of all the special keys, key sequences, and how their effects vary for the different windows.

Display the C source version of the sample file

Now that you can find your way around the debugger interface, you can get familiar with some of the debugger's more significant features. It's time to load some C code.

Display the contents of a C source file:

file sample.c ⮐

This opens a FILE window that displays the contents of the file sample.c (sample.c was one of the files that contributed to making the sample object file). You can always tell which file you're displaying by the label in the FILE window. Right now, the label should say FILE: sample.c.

Execute some code

Let's run some code—not the whole program, just a portion of it.

Execute a portion of the sample program:

go main ⮐

You've just executed your program up to the point where main() is declared. Notice how the display has changed:

❑ The current PC is highlighted in both the DISASSEMBLY and FILE windows.

❑ The addresses and object code of the first four statements in the DISASSEMBLY window are highlighted; this is because all four statements are associated with the current C statement (line 53 in the FILE window).

❑ The CALLS window, which tracks functions as they're called, now shows that the current function is main().

❑ The values of the program counter (PC), the stack pointer (SP), and possibly some additional registers are highlighted in the CPU window because they were changed by program execution.

Note: If you're using the emulator and the program doesn't stop after several seconds, press ⒺⓈⒸ and refer to Section A.4 (page A-7).

Become familiar with the three debugging modes

The debugger has three basic debugging modes:

❑ **Mixed mode** shows both disassembly and C code at the same time.

❑ **Auto mode** shows disassembly or C code, depending on what part of your program happens to be running.

❑ **Assembly mode** shows only the disassembly, no C code, even if you're executing C code.

When you opened the FILE window in a previous step, the debugger switched to mixed mode; you should be in mixed mode now. (You can tell that you're in mixed mode if both the FILE and DISASSEMBLY windows are displayed.)

The following steps show you how to switch debugging modes.

Use the **MoDe** menu to select assembly mode:

1) Look at the top of the display: the first line shows a row of pull-down menu selections.

2) Point to the word MoDe on the menu bar.

3) Press the left mouse button, but don't release it; drag the mouse downward until Asm (the second entry) is highlighted.

4) Release the button.

This switches to assembly mode. You should see the DISASSEMBLY window, but not the FILE window.

Switch to auto mode:

1) Type ⌈ALT⌉⌈D⌉. This displays and freezes the MoDe menu.

2) Now select C(auto). Choose one of these methods for doing this:

❑ Press the arrow keys to move up/down through the menu; when C(auto) is highlighted, press ⌈↵⌉.

❑ Type C ⌈↵⌉ .

❑ Point the mouse cursor at C(auto), then press and release the left mouse button.

You should be in auto mode now, and you should see the FILE window but not the DISASSEMBLY window (because your program is in C code). Auto mode automatically switches between an assembly or a C display, depending on where you are in your program. Here's a demonstration of that:

Run to a point in your program that executes assembly language code:

go meminit 🖳

You're still in auto mode, but you should now see the DISASSEMBLY window. The current PC should be at the statement that defines the meminit label.

Try This: You can also switch modes by typing one of these commands:

asm switches to assembly-only mode
c switches to auto mode
mix switches to mixed mode

Switch back to mixed mode.
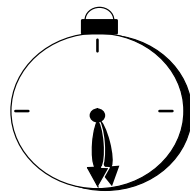
Halfway Point

You've finished the first half of the tutorial and the first set of lessons.

If you're lucky enough to be going to lunch or going home at this point, you may want to close the debugger down. To do this, just type **QUIT** 🖳. When you come back, reinvoke the debugger and load the sample program (page v). Then turn to page xvi and continue with the second set of lessons.

Still here? Turn the page.

Open another text file, then redisplay a C source file

In addition to what you already know about the FILE window and the FILE command, you should also know that:

❑ You can display any text file in the FILE window.

❑ If you enter any command that requires the debugger to display a C source file, it automatically displays that code in the FILE window (regardless of whether the window is open or not and regardless of what is already displayed in the FILE window).

Display a file that isn't a C source file:

file init.cmd 🖅

This replaces sample.c in the FILE window with the init.cmd file (init.cmd is a batch file that comes with the debugger).

Remember, you can tell which file you're displaying by the label in the FILE window. Right now, the label should say FILE: init.cmd.

Redisplay another C source file (sample1.c):

func call 🖅

Now the FILE window label should say FILE: sample1.c because the call() function is in sample1.c.

Use the basic RUN command

The debugger provides you with several ways of running code, but it has one basic run command.

Run your entire program:

run 🔁

Entered this way, the command basically means "run forever". You may not have that much time!

This isn't very exciting: halt program execution:

ESC

Set some breakpoints

When you halted execution in the previous step, you should have seen changes in the display similar to the changes you saw when you entered *go main* earlier in the tutorial. When you pressed ESC , you had little control over where the program stopped. Knowing that information changed was nice, but what part of the program affected the information?

This information would be much more useful if you picked an explicit stopping point before running the program. Then, when the information changed, you'd have a better understanding of what caused the changes. You can stop program execution in this way by setting *breakpoints*.

Important! This lesson assumes that you're displaying the contents of sample.c in the FILE window. If you aren't, enter:

file sample.c 🔁

Set a breakpoint and run your program:

1) Scroll to line 58 in the FILE window (the meminit() statement) and set a breakpoint at that line:

   a) Point the mouse cursor at the statement on line 58.

   b) Click the left mouse button. *Notice how the line is highlighted; this identifies a breakpointed statement.*

2) Reset the program entry point:

   restart [⏎]

3) Enter the run command:

   run [⏎]   *Program execution halts at the breakpoint*

Once again, you should see that some statements are highlighted in the CPU window, showing that they were changed by program execution. But this time, you know that the changes were caused by code from the beginning of the program to line 58 in the FILE window.

Clear the breakpoint:

1) Point the mouse cursor at the statement on line 58. (It should still be highlighted from setting the breakpoint.)

2) Click the left mouse button. *The line is no longer highlighted.*

Benchmark a section of code **(emulator only)**

If you're using the '34020 emulator, you can use breakpoints to help you benchmark a section of code. This means that you'll ask the debugger to count the number of CPU clock cycles that are consumed by a certain portion of code.

Benchmark some code:

1) In sample.c (displayed in the FILE window), set two breakpoints: one at line 58 (the meminit() statement) and one at line 66 (the for(;;); statement).

2) Reset the program entry point:

   restart ⏎

3) Enter the run command:

   run ⏎   *This runs to the first breakpoint*

4) Enter the runb command:

   runb ⏎   *This runs to the second breakpoint*
                                            (this may take several seconds)

5) Now use the ? command to examine the contents of the CLK pseudo-register:

   ? clk ⏎

---

emulator only

The debugger now shows a number in the display area; this is the number of CPU clock cycles consumed by the portion of code between the two breakpointed C statements.

Important! The value in the CLK pseudoregister is valid *only* when you execute the RUNB command and when that execution is halted on breakpointed statements.

Delete both breakpoints:

br ⏎   *The BR (breakpoint reset) command deletes*
                                    all breakpoints that were set

---

Watch some values and single-step through code

Now you know how to update the display without running your entire program; you can set breakpoints to obtain information at specific points in your program. But what if you want to update the display after each statement? No, you don't have to set a breakpoint at every statement—you can use single-step execution.

For this lesson, you have to be at a specific point in the program—let's go there before we do anything else.

Set up for single-step example:

restart ⏎
go main ⏎

The debugger has another type of window called a WATCH window that's very useful in combination with single-step execution. What's a WATCH window for? Suppose you are interested in only a few specific register values, not *all* of the registers shown in the CPU window. Or suppose you are interested in a particular memory location or in the value of some variable. You can observe these data items in a WATCH window.

Set up the WATCH window before you start the single-step execution.

Open a WATCH window:

wa a15, Stack Pointer ⏎
wa pc ⏎
wa *0xffc36440, Call: ⏎
wa i ⏎

You may have noticed that the WA (watch add) command can have one or two parameters. The first parameter is the item that you're watching. The second parameter is an optional label.

If the WATCH window isn't wide enough to display the PC value, resize the window.

Now try out the single-step commands. **Hint:** Watch the PC in the FILE and DISASSEMBLY windows; watch the value of i in the WATCH window.

Single-step through the sample program:

step 50 🖉

Try This: Notice that the step command single-stepped each assembly language statement (in fact, you single-stepped through 50 assembly language statements). Did you also notice that the FILE window displayed the source for the call() function when it was called? The debugger supports more single-step commands that have a slightly different flavor.

❏ For example, if you enter:

cstep 50 🖉

you'll single-step 50 *C statements,* not assembly language statements (notice how the PC "jumps" in the DISASSEMBLY window).

❏ Reset the program entry point and run to main().

restart 🖉
go main 🖉

Now enter the NEXT command, as shown below. You'll be single-stepping 50 assembly language statements, *but the FILE window doesn't display the source for the call() function when call() is executed.*

next 50 🖉

(There's also a CNEXT command that "nexts" in terms of C statements.)

Run code conditionally

Take a look at the code following the meminit() statement; this code is doing a lot of work with a variable named i. You may want to check the value of i at specific points instead of after each statement. To do this, you set breakpoints at the statements you're interested in and then initiate a conditional run.

First, clear out the WATCH window so that you won't be distracted by any superfluous data items.

Delete the first three data items from the WATCH window (don't watch them anymore):

```
wd 3  ⏎
wd 1  ⏎
wd 1  ⏎
```

---

i was the fourth item added to the WATCH window in the previous tutorial step, and it should now be the only remaining item in the window.

Set up for the conditional run examples

1) Set breakpoints at lines 59 and 61.

2) Reset the program entry point:

   ```
   restart  ⏎
   ```

3) Run the first part of the program

   ```
   go main  ⏎
   ```

4) Reset the value of i:

   ```
   ?i=0  ⏎
   ```

Now initiate the conditional run:

```
run i<100  ⏎
```

This causes the debugger to run through the for loop as long as the value of i is less than 100. Each time the debugger encounters the breakpoints in the loop, it updates the value of i in the WATCH window.

When the conditional run completes, close the WATCH window.

Close the WATCH window:

wr ⌐⟋¬

WHATIS that?

At some point, you might like to obtain some information about the types of data in your C program. Maybe things won't be working quite the way you'd planned, and you'll find yourself saying something like "... but isn't that supposed to point to an integer?" Here's how you can check on this kind of information—be sure to watch the COMMAND window display area as you enter these commands.

> Use the WHATIS command to find the types of some of the variables de-
> clared in the sample program:
>
> **whatis genum** ⏎
> ```
>      enum yyy genum;
> ```
> *genum is an enumerated type*
> **whatis tiny6** ⏎
> ```
>      struct {
>          int u;
>          int v;
>          int x;
>          int y;
>          int z;
>      } tiny6;
> ```
> *tiny6 is a structure*
> **whatis call** ⏎
> ```
>      int call();
> ```
> *call is a function that returns an integer*
> **whatis s** ⏎
> ```
>      short s;
> ```
> *s is a short unsigned integer*
> **whatis bbb** ⏎
> ```
>      struct bbb {
>          int a;
>          int b;
> ```
> *bbb is a very long structure*
> *Press* ⎡ESC⎤ *to halt long listings*

Clear the COMMAND window display area

After displaying all of these types, you may want to clean out the display area.
This is easy to do.

> Clear the COMMAND window display area:
>
> **cls** ⏎

Try This: Here are examples of two more DOS-like commands that you can
use in the debugger environment:

```
cd ..                         Change back to the main directory
dir                           Show a listing of the current directory
```

Display the contents of an aggregate data type

The WATCH window is convenient for watching single, or *scalar*, values. When
you're debugging a C program, though, you may need to observe values that
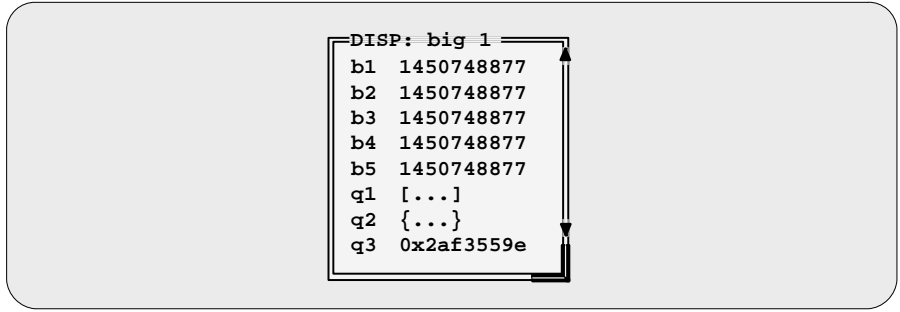
aren't scalar; for example, you might need to observe the effects of program execution on an array. The debugger provides another type of window called a DISP window where you can display the individual members of an array or structure.

Show another structure in a DISP window:

disp big1 ⟨✐⟩

Now you should see a display like the one below. The newly opened DISP window becomes the active window. Like the FILE window, you can always tell what's being displayed because of the way the DISP window is labeled. Right now, it should say DISP: big1.

```
┌─DISP: big 1 ═══╗▲
│b1  1450748877 ║
│b2  1450748877 ║
│b3  1450748877 ║
│b4  1450748877 ║
│b5  1450748877 ║
│q1  [...]      ║
│q2  {...}      ║
│q3  0x2af3559e ║▼
└───────────────╝
```

lesson continues on the next page →

lesson continues on the next page →

❏ Members b1, b2, b3, b4, and b5 are ints; you can tell because they're displayed as integers (shown as plain numbers without prefixes).

❏ Member q1 is an array; you can tell because q1 shows [. . .] instead of a value.

❏ Member q2 is another structure; you can tell because q2 shows {. . .} instead of a value.

❏ Member q3 is a pointer; you can tell because it is displayed as a hexadecimal address (indicated by a 0x prefix) instead of an integer value.

If a member of a structure or an array is itself a structure or an array, or even a pointer, you can display its members (or the data it points to) in additional DISP windows (referred to as the original DISP window's *children*).

Display what q3 is pointing to:

🖱   1)   Point at the address displayed next to the q3 label in big1's display.

🖱   2)   Click the left mouse button.

This opens a second DISP window, labeled `DISP: big1.q3`, that shows what q3 is pointing to (it's pointing to another structure). Close this DISP window by pressing `F4`.

Display array q1 in another DISP window:

🖱   1)   Point at the [. . .] displayed next to the q1 label in big1's display.

🖱   2)   Click the left mouse button.

This opens another DISP window labeled `DISP: big1.q1`.

Important! q1 is actually a 2-member array of structures. To view the two different structures, use `CONTROL` `PAGE DOWN` and `CONTROL` `PAGE UP`. (Look at the name of this DISP window when you're switching.)

Try This: Display structure q2 in another DISP window.

1)   Close the additional DISP windows or move them out of the way so that you can clearly see the original DISP window that you opened to display big1.

2)   Make big1's DISP window the active window.

⬇⬆   3)   Use these arrow keys to move the field cursor (_) through the list of big1's members until the cursor points to q2.

`F9`   4)   Now press `F9`.

This opens a window named `DISP: big1.q2`.

Close all of the DISP windows:

1) Make `DISP: big1` the active window.

2) Press `F4`.

When you close the main DISP window, the debugger closes all of its children as well.

Display data in another format

Usually, when you add an item to the WATCH window or open a DISP window, the data is shown in its *natural format*. This means that ints are shown as integers, floats are shown as floating-point values, etc. Occasionally, you may wish to view data in a different format. This can be especially important if you want to show memory or register contents in another format.

One way to display data in another format is through casting (which is part of the C language). In the expression below, the *(float *) portion of the expression tells the debugger to treat address 0x1000 as type float (exponential floating-point format).

Display memory contents in floating-point format:

```
disp *(float *)0x1000  ⏎
```

This opens a DISP window to show memory contents in an array format. The "array" member identifiers don't necessarily correspond to actual addresses—they're relative to the first address you request with the DISP command. In this case, the item displayed as item [0] is the contents of address 0x1000—*it isn't memory location 0*. Note that you can scroll through the memory displayed in the DISP window; item [1] is at 0x1010, item [–1] is at 0x0FE0.

You can also change display formats according to data type. This affects all data of a specific C data type.

Change display formats according to data types by using the SETF (set format) command:

1) For comparison, watch the following variables. Their C data types are listed on the right.

```
wa str.f1  ⏎
wa longstr.A  ⏎
wa d  ⏎
wa ac[1]  ⏎
```
          Type int
          Type float
          Type double
          Type char

2) You can list all the data types and their current display formats:

```
setf  ⏎
```

---

3) Now display the following data types with new formats:

`setf int, c` ⌨   *Ints as characters*
`setf float, o` ⌨   *Floats as octal integers*
`setf double, x` ⌨   *Doubles as hex integers*
`setf char, d` ⌨   *Chars as decimal integers*

4) List the data types to display formats again; note the changes in the display:

`setf` ⌨

5) Add the variables to the WATCH window again; use labels to identify the additions:

`wa str.f1, NEWstr.f1` ⌨
`wa longstr.A, NEWlongstr.A` ⌨
`wa d, NEWd` ⌨
`wa ac[1], NEWac[1]` ⌨

Notice the differences in the display formats between the first versions you added and these new versions.

6) Now reset all data types back to their defaults:

`setf *` ⌨

A third way to display data in another format is to use the DISP, ?, MEM, or WA command with an optional parameter that identifies the new display format. The following examples are for ? and WA—DISP and MEM work similarly.

Use display formats with the ? and WA commands:

1) Evaluate a variable and display it as a character:

`? small.ra[1],c` ⌨

2) Add a variable to the watch window and display it as an octal integer:

`wa longstr.a,,o` ⌨

(Notice that because no label was used with WA, an extra comma was inserted—otherwise, the o parameter would have been interpreted as a label.)

To get ready for the next step, close the DISP and WATCH windows.

Change some values

You can edit the values displayed in the MEMORY, CPU, WATCH, and DISP windows.

Change a value in memory:

    1) Move or close the WATCH window if it's obscuring the MEMORY window; then, display memory beginning with address 0x1000:

        mem 0x1000 🔎

    2) Point to the contents of memory location 0x1000.

    3) Click the left mouse button. This highlights the field to identify it as the field that will be edited.

    4) Type 0000.

    5) Press 🔎 to enter the new value.

    6) Press ⌨ESC⌨ to conclude editing.

Try This: Here's another method for editing data. This method lets you edit a few more values at once.

1) Make the CPU window the active window:

    win CPU ⏎

⬆⬇ 2) Press the arrow keys until the field cursor (_) points to the PC contents.

F9 3) Press F9 .

4) Type ffc000000.

⬇ 5) Press ⬇ **twice**. You should now be pointing at the contents of register A2.

6) Type ffff.

⏎ 7) Press ⏎ to enter the new value.

ESC 8) Press ESC to conclude editing.

Define a memory map

You can set up a memory map to tell the debugger which areas of memory it can and can't access. This is called *memory mapping*. By default, memory mapping is disabled, which means that the debugger assumes that all of memory is available for its use. For the purposes of the sample program, that's fine (which is why this lesson was saved for next-to-last).

Memory mapping is a simple process of identifying the memory space in your application and identifying whether the spaces are read/write, protected, etc. Here's an example.

Enable memory mapping:

map on [↵]

Look at the DISASSEMBLY and MEMORY windows. Mapping is enabled, but the actual memory map hasn't been defined yet. So, all of the address values in the MEMORY are highlighted as invalid (on color monitors, the default method for indicating invalid memory contents is to show them in red). Because memory contents is now invalid, the disassembly of memory in the DISASSEMBLY window no longer shows assembly language code—each line says *Invalid address*.

Define a memory map:

1) Use the MA (memory add) command to map a block of program memory:

   ma 0xffc00000,0x0010000,RAM [↵]

   Program memory is now valid — the DISASSEMBLY window displays assembly language code again.

2) Add in a block of memory for '340 I/O registers:

   ma 0xc0000000,0x10000000,RAM [↵]

Try This: The debugger supports a memory list command that tells you how memory is currently mapped:

ml [↵]

Look in the COMMAND window display area — you'll see a listing of the two areas that you defined.

Define your own command string

If you find that you often enter a command with the same parameters, or often enter the same commands in sequence, you will find it helpful to have a shorthand method for entering these commands. The debugger provides an *aliasing* feature that allows you to do this.

This lesson shows you how you can define an alias to set up a memory map, defining the same map that was defined in the previous lesson.

Define an alias for setting up the memory map:

1) Use the ALIAS command to associate a nickname with the commands used for defining a memory map:

   **alias mymap,"mr;ma 0xffc00000,0x0010000,RAM;
                ma0xc0000000,0x10000000,RAM;ml"** 〔⏎〕

   (Note: Because of space constraints, the command is shown on two lines.)

2) Now, to use this memory map, just enter the alias name:

   mymap 〔⏎〕

   This is equivalent to entering the following four commands:

   ```
   mr
   ma 0xffc00000,0x0010000,RAM
   ma 0xc0000000,0x10000000,RAM
   ml
   ```

Close the debugger

This is the end of the tutorial — close the debugger.

Close the debugger and return to DOS:

quit 〔⏎〕

Overview of a Code
Development and Debugging System

The '340 C source debugger is an advanced software interface that helps you to develop, test, and refine '340 C programs (compiled with the '340 optimizing ANSI C compiler) and assembly language programs. This chapter provides an overview of the C source debugger and describes the '340 code development environment.

Synopsis	**Topic**	**Page**

5.1 Description of the '340 C Source Debugger

The '340 C source debugger improves productivity by allowing you to debug a program in the language it was written in. You can choose to debug your programs in C, assembly language, or both. And, unlike many other debuggers, the '340 debugger's higher level features are available even when you're debugging assembly language code.

The debugger is easy to learn and use. Its friendly window-, mouse-, and menu-oriented interface reduces learning time and eliminates the need to memorize complex commands. The debugger's customizable displays and flexible command entry let you develop a debugging environment that suits your needs—you won't be locked into a rigid environment. A shortened learning curve and increased productivity reduce the software development cycle, so you'll get to market faster.

Figure 5–1 identifies several features of the debugger display.

Figure 5–1. The Debugger Display

Key features of the debugger

❏ **Multilevel debugging**.The debugger allows you to debug both C and assembly language code. If you're debugging a C program, you can choose to view just the C source, the disassembly of the object code created from the C source, or both. You can also use the debugger as an assembly language debugger.

❏ **Fully configurable, state-of-the-art, window-oriented interface.** The C source debugger separates code, data, and commands into manageable portions. Use any of the default displays. Or, select the windows you want to display, size them, and move them where you want them.

❏ **Comprehensive data displays.** You can easily create windows for displaying *and editing* the values of variables, arrays, structures, pointers—any kind of data — in their natural format (float*,* int*,* char*,* enum*,* or pointer*).* You can even display entire linked lists.

```
┌DISP: str ─────                    ┌WATCH ──────
 a   123              ▲             1: str.a 0
 b   0                              2: F0 1.000000e
 c   75435┌DISP: *str.f3 ──         3: color GREEN
 f1  3    │ a   8327         ▲
 f2  6    │ b   666
 f3  0x00f│ c   87213┌DISP: *str.f3->f3 ─┐
 f4  [...]│ f1  45   │ a   75              ▲
          │ f2  27   │ b   3212
          │ f3  0x00f│ c   782
          │ f4  [...]│ f1  7
                     │ f2  9
                     │ f3  0x00f000a      ▼
                     │ f4  [...]
```
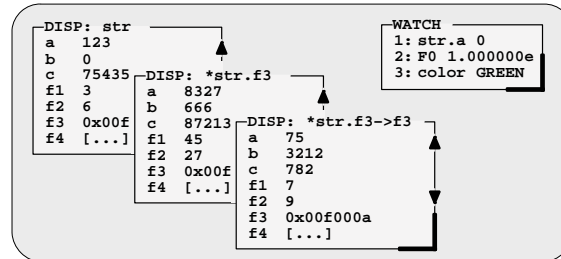
❏ **On-screen editing.** Change any data value displayed in any window —just point the mouse, click, and type.

❏ **Continuous update.** The debugger continuously updates information on the screen, highlighting changed values.

❏ **Powerful command set.** Unlike many other debugging systems, this debugger doesn't force you to learn a large, intricate command set. The '340 C source debugger supports a small but powerful command set that makes full use of C expressions. One debugger command performs actions that would take several commands in another system.

❏ **Flexible command entry.** There are a variety of ways to enter commands. You can type commands or use a mouse, function keys, or the pulldown menus; choose the method that you like best. Want to re-enter a command? No need to retype it—simply use the command history.

❏ **Create your own debugger.** The debugger display is completely configurable, allowing you to create the interface that is best suited for your use.

   ■ If you're using a color display, you can change the colors of any area on the screen.

   ■ You can change the physical appearance of display features such as window borders.

   ■ You can interactively set the size and position of windows in the display.

   Create and save as many custom configurations as you like, or use the defaults. Use the debugger with a color display or a black-and-white display. A color display is preferable; the various types of information on the display are easier to distinguish when they are highlighted with color.

❏ **Variety of screen sizes.** The debugger's default configuration is set up for a typical PC display, with 25 lines by 80 characters. If you use a sophisticated graphics card, you can take advantage of the debugger's additional screen sizes. A larger screen size allows you to display more information and provides you with more screen space for organizing the display—bringing the benefits of workstation displays to your PC.

❏ **Plus, all the standard features you expect in a world-class debugger.** The debugger provides you with complete control over program execution with features like conditional execution and single-stepping (including single-stepping into or over function calls). You can set or clear a breakpoint with a click of the mouse or by typing commands. You can define a memory map that identifies the portions of target memory that the debugger can access. You can choose to load only the symbol table portion of an object file to work with systems that have code in ROM. The debugger can execute commands from a batch file, providing you with an easy method for entering often-used command sequences.

5.2 **Developing Code for the '340**

The '340 is supported by a complete set of hardware and software development tools, including a C compiler, assembler, and linker. Figure 5–2 illustrates the '340 code development flow. The figure highlights the most common paths of software development; the other portions are optional.

Figure 5–2. '340 Software Development Flow
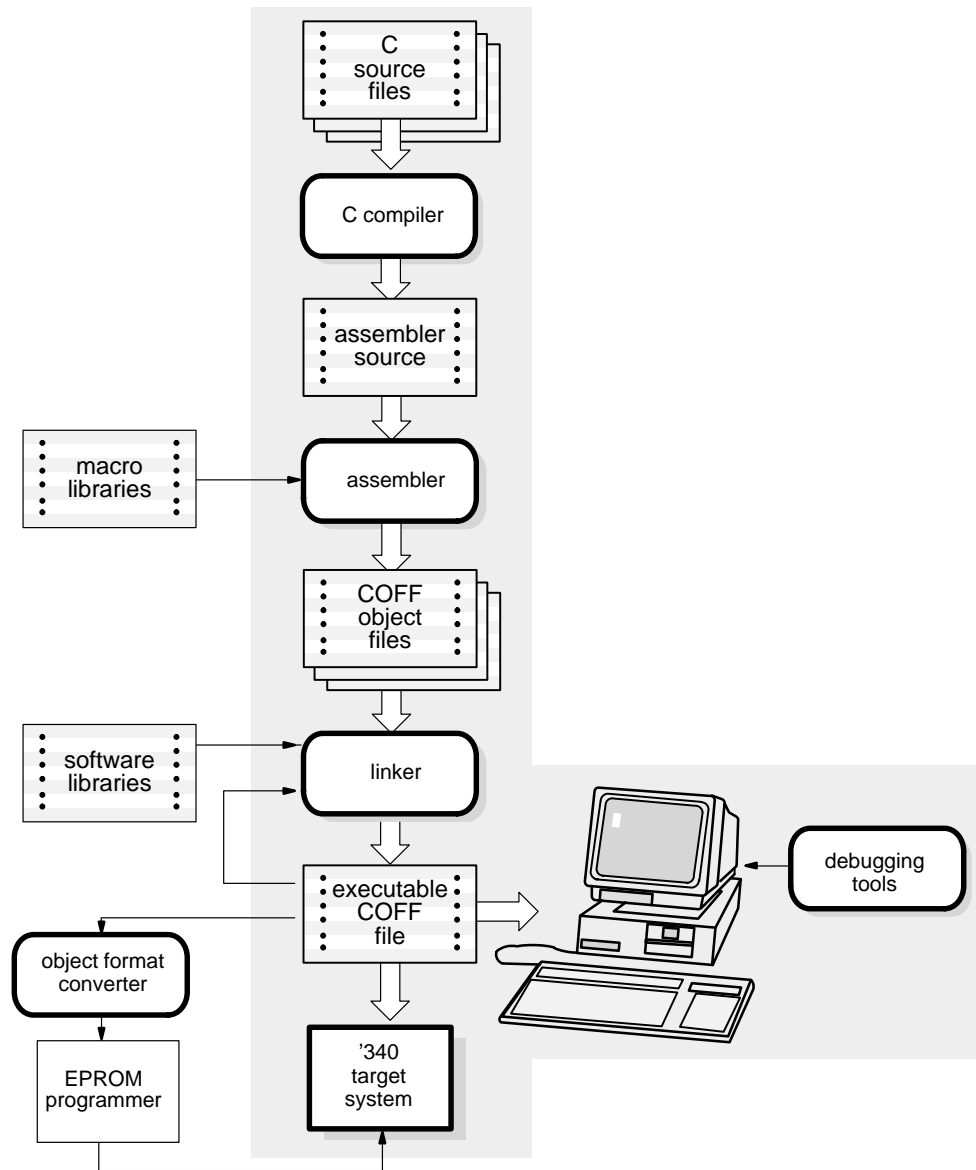
These tools use common object file format (COFF), which encourages modular programming. COFF allows you to divide your code into logical blocks, define your system's memory map, and then link code into specific memory areas. COFF also provides rich support for source-level debugging.

The following list describes the tools shown in Figure 5–2.

C compiler

The '340 **optimizing ANSI C compiler** is a full-featured optimizing compiler that translates standard ANSI C programs into '340 assembly language source. Key characteristics include:

❏ *Standard ANSI C.* The ANSI standard is a precise definition of the C language, agreed upon by the C community. The standard encompasses most of the recent extensions to C. To an increasing degree, ANSI conformance is a requirement for C compilers.

❏ *Optimization.* The compiler uses several advanced techniques for generating efficient, compact code from C source.

❏ *Assembly language output.* The compiler generates assembly language source that you can inspect (and modify, if desired).

❏ *ANSI standard runtime support.* The compiler package comes with a complete runtime library that conforms to the ANSI C library standard. The library includes functions for string manipulation, dynamic memory allocation, data conversion, timekeeping, trigonometry, exponential, and hyperbolic functions. Functions for I/O and signal handling are not included, because they are application specific.

❏ *Flexible assembly language interface.* The compiler has straightforward calling conventions, allowing you to easily write assembly and C functions that call each other.

❏ *Shell program.* The compiler package includes a shell program that enables you to compile, assemble, and link programs in a single step.

❏ *Source interlist utility.* The compiler package includes a utility that interlists your original C source statements into the assembly language output of the compiler. This utility provides you with an easy method for inspecting the assembly code generated for each C statement.

assembler

The **assembler** translates '340 assembly language source files into machine language object files.

archiver

The **archiver** allows you to collect a group of files into a library. It also allows you to modify a library by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is to build a library of object modules. Several object libraries and a source library are included with the C compiler.

linker

The **linker** combines object files into a single, executable object module. As the linker creates the executable module, it performs relocation and resolves external references. The linker is a tool that allows you to define your system's memory map and to associate blocks of code with defined memory areas.

debugging
tools

The main purpose of the development process is to produce a module that can be executed in a **'340 target system.** You can use one of a variety of **debugging tools** to refine and correct your code. Each uses the '340 debugger as a software interface:

❑ The emulator version of the debugger supports the TI realtime emulator for the '34020.

❑ The development board version of the debugger supports an entire class of '340-based PC development boards that use TIGA and a TIGA communication driver. TI has two such boards: the '34010 TIGA development board and the '34020 software development board.

object
format
converter

An **object format converter** is also available; it converts a COFF object file into an Intel, Tektronix, or TI-tagged object-format file that can be downloaded to an EPROM programmer.

5.3 Preparing Your Program for Debugging

This section describes the steps that you must go through when preparing a '340 program and discusses constraints that apply to development board programs.

Assembling and/or compiling your program

Figure 5–3 illustrates the steps you must go through to prepare a program for debugging.

Figure 5–3. Steps You Go Through to Prepare a Program
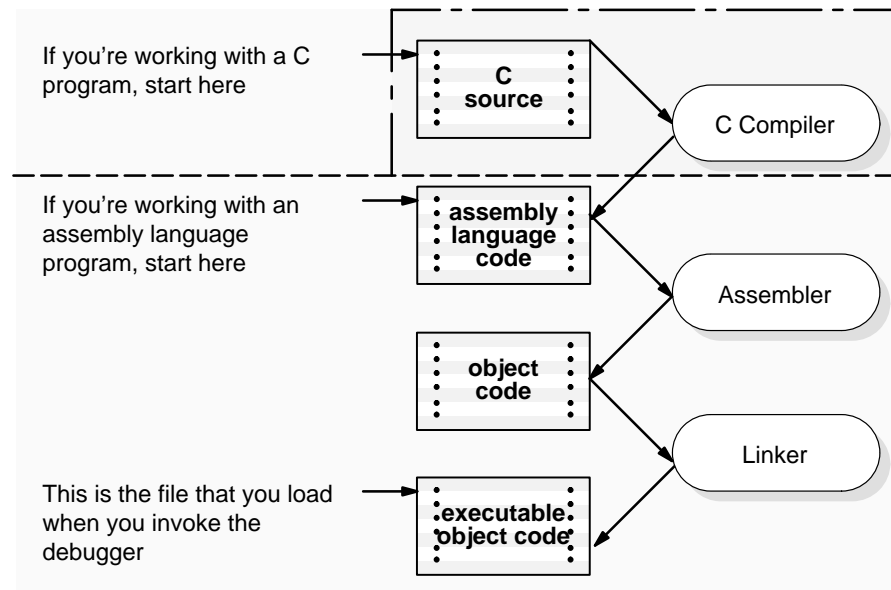
If you're preparing to debug a C program. . .	1) Compile the program; **use the –g option.** Use the –mc option for '34082 support. Use the –mf option for IEEE floating-point support without the '34082 support.
	2) Assemble the resulting assembly language program.
	3) Link the resulting object file.
	This produces an object file that you can load into the debugger.
If you're preparing to debug an assembly language program. . .	1) Assemble the assembly language source file.
	2) Link the resulting object file.
	This produces an object file that you can load into the debugger.

You can compile, assemble, and link a program by invoking the compiler, assembler, and linker in separate steps; or, you can perform all three actions in a single step by using the GSPCL shell program. The *TMS340 Code Generation Tools User's Guide* contains complete instructions for invoking the tools individually and for using the shell program.

For your convenience, here's the basic command for invoking the shell program when preparing a program for debugging:

gspcl [–*options*] **–g** [*filenames*] [**–v***nn*] [**–mc** | **–mf**] [**–z** [*link options*]]

gspcl	is the command that invokes the compiler and assembler.
options	affect the way the shell processes input files.
filenames	are one or more C source files, assembly language source files, or object files. Filenames are not case sensitive.
–g	is an option that tells the C compiler to produce symbolic debugging information. When preparing a C program for debugging, you must use the –g option.
–mc	provides '34082 support. If you plan to use the debugger's –mc option, then you must also use the –mc option when compiling your code.
–mf	provides IEEE support (without '34082 support). If you plan to use the debugger's –mf option, then you must also use the –mf option when compiling your code.
–v	is an option that tells the tools to produce code for either the '34010 or '34020; the *nn* is a number that identifies the correct device:

       –v10   creates object code for the '34010

       –v20   creates object code for the '34020

       By default, the tools expect '34020 source and create '34020 object. So, if you are debugging a '34020 program, it's not necessary to use the –v option. However, if you are debugging a '34010 program, you *must* use the –v option.

–z	is an option that invokes the linker. After compiling/assembling your programs, you can invoke the linker in a separate step. If you want the shell to automatically invoke the linker, however, use –z.
link options	affect the way the linker processes input files; use these options only when you use –z.

Options and filenames can be specified in any order on the command line, but if you use –z, it must follow all C/assembly language source filenames and compiler options.

The shell identifies a file's type by the filename's extension.

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

Program constraints for development board applications

When preparing a '340 program for debugging on a development board, observe the following constraints:

❏ Your program should not modify these vectors or TRAP instructions:

Vector address	TRAP instruction	
0xFFFF FC40	TRAP 29	
	Used for debugger breakpoints	
0xFFFF FBE0	TRAP 32	**('34020 only)**
	Used for debugger single stepping	

❏ Your program should not rely on accessing previous stack contents that were popped off the stack.

❏ **34020 only:**

■ Your program should not use the single-step interrupt.

■ When an interrupt is serviced, the program counter and status register values are pushed onto the stack. If a RUN command is executing when the interrupt occurs, the single-step bit may be set in the status register value that is pushed on the stack. The interrupt service routine should not modify this bit.

5.4 Debugging '340 Programs

Debugging a program is a multiple-step process. These steps are described below, with references to parts of this book that will help you accomplish each step.

Step 1

Prepare a C program or assembly language program for debugging.

See Section 5.3, *Preparing a Program for Debugging*, page viii.

Step 2

Ensure that the debugger has a valid memory map.

See Chapter 8, *Defining a Memory Map*

Step 3

Load the program's object file.

See Section 9.3, *Loading Object Code*, page 9-8.

Step 4

Run the loaded file. You can run the entire program, run parts of the program, or single-step through the program.

See *Running Your Programs* on page 9–10.

Step 5

Stop the program at critical points and examine important information.

See Chapter 12, *Using Breakpoints,* and Chapter 11, *Managing Data.*

Step 6

If you find a mistake in your source code, exit the debugger, edit your source file, and return to step 1.

The Debugger Display

The '340 C source debugger has a window-oriented display. This chapter shows what windows can look like and describes the basic types of windows that you'll use.

Synopsis		**Topic**
Page		

6.1   Debugging Modes and Default Displays

The debugger has three debugging modes:

❏ Auto mode
❏ Assembly mode
❏ Mixed mode

Each mode changes the debugger display by adding or hiding specific windows. Some windows, such as the COMMAND window, may be present in all modes. The following figures show the default displays for these modes and show the windows that the debugger automatically displays for these modes. In addition to the default windows shown in these illustrations, you can also display DISP windows and the WATCH window (see Section 6.2, page vii).

Auto mode

In **auto mode**, the debugger automatically displays whatever type of code is currently running — assembly language or C. This is the default mode; when you first invoke the debugger, you'll see a display similar to Figure 6–1. Auto mode has two types of displays:

❏ When the debugger is running assembly language code, you'll see an assembly display similar to the one in Figure 6–1. The DISASSEMBLY window displays the reverse assembly of memory contents.

Figure 6–1. Typical Assembly Display (for Auto Mode and Assembly Mode)

```
 Load    Brea    Watch    Memory    Color    MoDe    Run=F5    Step=F8    Next=F10
┌─DISASSEMBLY─────────────────────────────────────────────────┐    ┌─CPU───────┐
│ ffc00850 0740   c_int00:    SETF    32,0,0                 ▲ │    │PC   ffc00850▲│
│ ffc00860 09e9               MOVI    -3972128,A9              │    │ST   00000010 │
│ ffc00890 4d2d               MOVE    A9,A13                   │    │A0   00000000 │
│ ffc008a0 4d2e               MOVE    A9,A14                   │    │A1   00000000 │
│ ffc008b0 09e9               MOVI    -3908192,A9              │    │A2   00000000 │
│ ffc008e0 4d2f               MOVE    A9,SP                    │    │A3   c0000010 │
│ ffc008f0 09ea               MOVI    -1,A10                   │    │A4   00000000 │
│ ffc00920 0b4a               CMPI    -1,A10                   │    │A5   00000000 │
│ ffc00940 ca05               JRZ     0ffc009a0h               │    │A6   00000000 │
│ ffc00950 a3cf               MOVE    A14,*-SP,1               │    │A7   00000040 │
│ ffc00960 934e               MOVE    A10,*A14+,1              │    │A8   00000000 │
│ ffc00970 0d5f               CALLA   var_init                 │    │A9   ffc45da0 │
│ ffc009a0 0d5f               CALLA   main                     │    │A10  ffffffff │
│ ffc009d0 a3cf               MOVE    A14,*-SP,1             ▼ │    │A11  00001795 │
│ ffc009e0 182b               MOVK    1,A11                    │    │A12  00000000 │
└─────────────────────────────────────────────────────────────┘    │A13  ffc363e0▼│
                                                                     │A14  ffc363e0 │
                                                                     │A15  ffffa000 │
┌─COMMAND─────────────────────────┐  ┌─MEMORY──────────────────────────────────┐
│TMS340 Debugger Version 5.00    ▲│  │00000000  4000 4001 4002 4003 40004 4005 4006▲│
│Copyright (c) 1990, Texas In-    │  │00000070  4007 4008 4009 400a 400b 400c 400d │
│strume                           │  │000000e0  400e 400f 4010 4011 4012 4013 4014 │
│TMS340x0 Development Board        │  │00000150  4015 4016 4017 4018 4019 401a 401b │
│_                               ▼│  │000001c0  401c 401d 401e 401f 4020 4021 4022▼│
│>>>                              │  │00000230  4023 4024 4025 4026 4027 4028 4029 │
└─────────────────────────────────┘  └─────────────────────────────────────────┘
```

❑ When the debugger is running C code, you'll see a C display similar to the one in Figure 6–2. (This assumes that the debugger can find your C source file to display in the FILE window. If the debugger can't find your source, then it switches to mixed mode.)

Figure 6–2. Typical C Display (for Auto Mode Only)

```
 Load    Brea    Watch   Memory   Color   MoDe   Run=F5   Step=F8   Next=F10
         k
 FILE: sample.c
00044 short  aaas[10][5][2];
00045 char   ac[10];
00046 int    *pi;
00047 char   *xpc;
00048 struct zzz giant[100];
00049
00050 extern call();
00051 exter  meminit();
00052
00053 main()
00054 {
00055        register int i = 0;
00056        int j = 0; k = 0;
00057
0058         meminit();

 COMMAND                                            CALLS
TMS340 Debugger Version 5.00                        1: main()
Copyright (c) 1990, Texas Instruments Incorporated
TMS340x0 Development Board
go main

>>>
```

When you're running assembly language code, the debugger automatically displays windows as described for assembly mode.

When you're running C code, the debugger automatically displays the COMMAND, CALLS, and FILE windows. If you like, you can also open a WATCH, DISP, I/O, or FPU window.

Assembly mode

Assembly mode is for viewing assembly language programs only. In this mode, you'll see a display similar to the one shown in Figure 6–1. When you're in assembly mode, you'll always see the assembly display, regardless of whether C or assembly language is currently running.

Windows that are automatically displayed in assembly mode include the MEMORY window, the DISASSEMBLY of memory contents, the CPU register window, and the COMMAND window. If you like, you can also open a WATCH, DISP, I/O, or FPU window.

Mixed mode

Mixed mode is for viewing assembly language and C code at the same time. Figure 6–3 shows the default display for mixed mode.

Figure 6–3. Typical Mixed Display (for Mixed Mode Only)

```
 Load    Brea    Watch    Memory    Color    MoDe    Run=F5    Step=F8    Next=F10
  ┌DISASSEMBLY                                                        ┌CPU
  fffc0000 098f   main:       MMTM     SP,A7,A9,A11,FP              PC    fffc0000 ▲
  ffc00020 4dcd               MOVE     STK,FP                       ST    00000010
  ffc00030 0b0e               ADDI     64,STK                       A0    00000000
  ffc00050 93ee               MOVE     SP,*STK+,1                   A1    00000000
  ffc00060 4529               SUB      A9,A9                        A2    00000000
  ffc00070 832d               MOVE     A9,@j,1                      A3    c0000010
  ffc00080 b32d               MOVE     A9,@k,1                      A4    00000000
  ffc000a0 0d5f               CALLA    meminit                      A5    00000000
  ffc000d0 a3cf               MOVE     STK,*-SP,1                   A6    00000000
                                                                    A7    00000040
  ┌FILE: sample.c                                                   A8    00000000 ▼
  00047  char   *xpc;                                         ▲     A9    ffc45da0
  00048   struct zzz giant[100];                                    A10   ffffffff
  00049                                                            ┌CALLS
  00050  extern call();                                           1: main()
  00051  exter  meminit();
  00052
  00053  main()
  ┌COMMAND                           ┌MEMORY
  file sample.c                       00000000  4000 4001 4002 4003 40004 4005 4006 ▲
  go main                             00000070  4007 4008 4009 400a 400b 400c 400d
  mix                                 000000e0  400e 400f 4010 4011 4012 4013 4014 ▼
                                      00000150  4015 4016 4017 4018 4019 401a 401b
  >>>                                 000001c0  401c 401d 401e 401f 4020 4021 4022
```

In mixed mode, the debugger displays all windows that can be displayed in auto and assembly modes—regardless of whether you're currently running assembly language or C code. This is useful for finding bugs in C programs that exploit specific architectural features of the '340.

Restrictions associated with debugging modes

The assembly language code that the debugger shows you is the disassembly (reverse assembly) of memory's contents. If you load object code into memory, then the assembly language code is the disassembly of that object code. If you don't load an object file, then the disassembly won't be very useful.

Some commands are valid only in certain modes, especially if a command applies to a window that is visible only in certain modes. In this case, entering the command causes the debugger to switch to the mode that is appropriate for the command. This applies to these commands:

dasm	func	mem
calls	file	disp

6.2 Descriptions of the Different Kinds of Windows and Their Contents

The debugger can show several types of windows. This section lists the various types of windows and describes their characteristics.

Every window is identified by a name in its upper left corner. Each type of window serves a specific purpose and has unique characteristics. There are eight different windows, divided into three general categories:

❏ The **COMMAND window** provides an area for typing in commands and for displaying various types of information such as progress messages, error messages, or command output.

❏ **Code-display windows** are for displaying assembly language or C code. There are three code-display windows:

■ The DISASSEMBLY window displays the disassembly (assembly language version) of memory contents.

■ The FILE window displays any text file that you want to display; its main purpose, however, is to display C source code.

■ The CALLS window identifies the current function traceback (when C code is running).

❏ **Data-display windows** are for observing and modifying various types of data. There are six data-display windows:

■ The MEMORY window displays the contents of a range of memory.

■ The CPU window displays the contents of '340 registers.

■ The I/O window displays the contents of '340 I/O registers.

■ The FPU window displays the contents of '34082 registers.

■ A DISP window displays the contents of an aggregate type such as an array or structure, showing the values of the individual members. You can display up to 120 DISP windows at one time.

■ A WATCH window displays selected data such as variables, specific registers, or memory locations.

You can move or resize any of these windows; you can also edit any value in a data-display window. Before you can perform any of these actions, however, you must select the window you want to move, resize, or edit, and make it *the active window*. For more information about making a window active, see Section 6.4, *The Active Window*, on page xx.

The remainder of this section describes the individual windows.

COMMAND window

```
┌─COMMAND─────────────────────────────────────────────┐
display      │ TMS340 Debugger Version 5.00                    ▲ │
area         │ Copyright (c) 1990, Texas Instruments Incorporated │
             │ TMS34010 Rev 1                                    │
             │                                                   │
             │ Loading sample.out                              ▼ │
command      │ >>>  go main   ▐                                  │
line         └───────────────────────────────────────────────────┘

                              command line
                              cursor
```
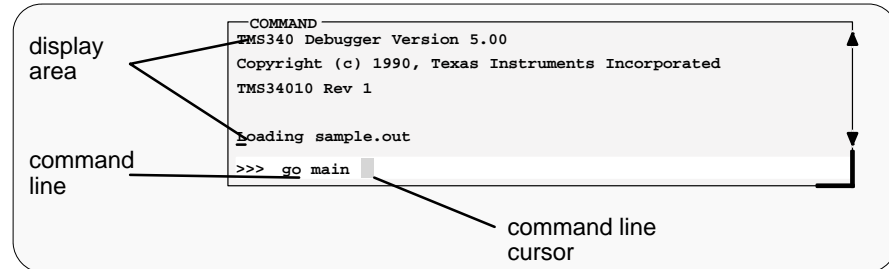
Purpose	Provides an area for entering commands
	Provides an area for echoing commands and displaying command output, errors, and messages
Editable?	Command line is editable; command output isn't
Modes	All modes
Created	Automatically
Affected by	Any command entered on the command line Any command displaying output in the display area Any input that creates an error

The COMMAND window has two parts:

❑ **Command line.** This is where you enter commands. When you want to enter a command, just type — no matter which window is active. The debugger keeps a list of the last 100 commands that you entered. You can select and re-enter commands from the list without retyping them.

❑ **Display area**. This area of the COMMAND window echoes the command that you entered, shows any output from the command, and displays debugger messages.

For more information about the COMMAND window and entering commands, refer to Chapter 7, *Entering and Using Commands*.

DISASSEMBLY window
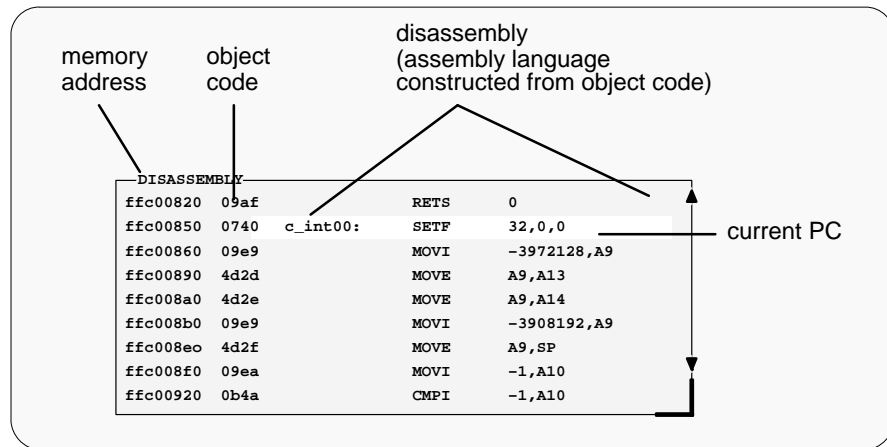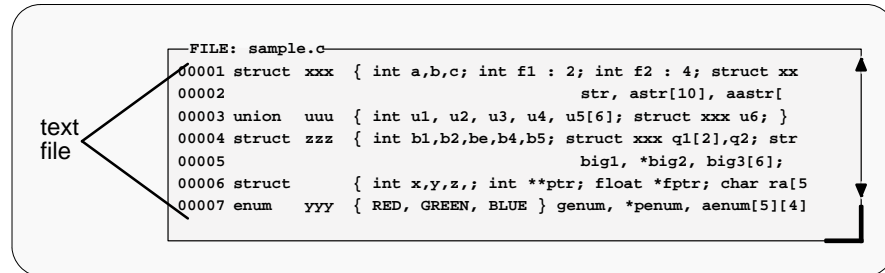
Purpose	Displays the disassembly (or reverse assembly) of memory contents
Editable?	No; pressing the edit key (F9) or the left mouse button sets a breakpoint on an assembly language statement
Modes	Auto (assembly display only), assembly, and mixed
Created	Automatically
Affected by	DASM and ADDR commands Breakpoint and run commands

Within the DISASSEMBLY window, the debugger highlights:

❏ The statement that the PC is pointing to (if that line is in the current display)
❏ Any breakpointed statements
❏ The address and object code fields for all statements associated with the current C statement, as shown below

FILE window

Purpose	Shows any text file you want to display
Editable?	No; if the FILE window displays C code, pressing the edit key (F9) or the left mouse button sets a breakpoint on a C statement
Modes	Auto (C display only) and mixed
Created	With the FILE command Automatically when you're in auto or mixed mode and your program begins executing C code
Affected by	FILE, FUNC, and ADDR commands Breakpoint and run commands

You can use the FILE command to display the contents of any file within the FILE window, but this window is especially useful for viewing C source files. Whenever you single-step a program or run a program and halt execution, the FILE window automatically displays the C source associated with the current point in your program. This overwrites any other file that may have been displayed in the window.

Within the FILE window, the debugger highlights:

❑ The statement that the PC is pointing to (if that line is in the current display)
❑ Any statements where you've set a breakpoint

CALLS window

order of functions called

current function is at top of list

names of functions called

```
CALLS
3: subx()
2: call()
1: main()
```
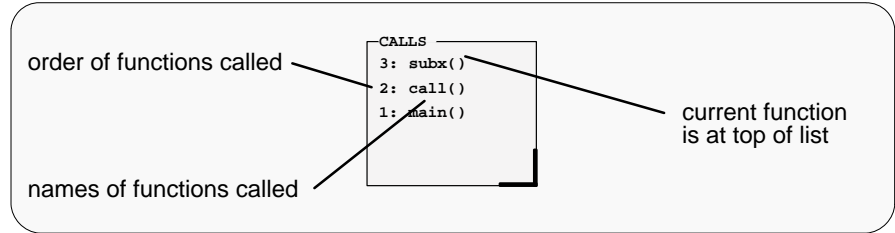
Purpose	Lists the function you're in, its caller, and its caller, etc., as long as each function is a C function
Editable?	No; pressing the edit key (F9) or the left mouse button changes the FILE display to show the source associated with the called function
Modes	Auto (C display only) and mixed
Created	Automatically when you're displaying C code With the CALLS command if you closed the window
Affected by	Run and single-step commands

The display in the CALLS window changes automatically to reflect the latest function call.

If you haven't run any code, then no functions have been called yet. You'll also see this if you're running code but are not currently running a C function.

```
CALLS
1: **UNKNOWN
```

In C programs, the first C function is main.

```
CALLS
1: main()
```

As your program runs, the contents of the CALLS window change to reflect the current routine that you're in and where the routine was called from. When you exit a routine, its name is popped from the CALLS list.

```
CALLS
2: xcall()
1: main()
```

```
CALLS
1: main()
```

If a function name is listed in the CALLS window, you can easily display the function in the FILE window:

↖ 1) Point the mouse cursor at the appropriate function name that is listed in the CALLS window.

⬚ 2) Click the left mouse button. This displays the selected function in the FILE window.

1) Make the CALLS window the active window (see Section 6.4, *The Active Window*, page xx).

⬚⬚ 2) Use the arrow keys to move up/down through the list of function names until the appropriate function is indicated.

F9 3) Press F9 . This displays the selected function in the FILE window.

You can close and reopen the CALLS window.

❑ Closing the window is a two-step process:

1) Make the CALLS window the active window.

2) Press F4

❑ To reopen the CALLS window after you've closed it, enter the CALLS command. The format for this command is:

calls

MEMORY window

```
   ┌MEMORY──────────┐
   │ 00000000   4000 │
   │ 00000010   4001 │
addresses│ 00000020   4002 │ data
   │ 00000030   4003 │
   │ 00000040   4004 │
   │ 00000050   4005 │
   └────────────────┘

   ┌MEMORY──────────────────────────────────────────┐
   │ 00000000   4000  4001  4002  4003  4004  4005 │
   │ 00000060   4006  4007  4008  4009  400a  400b │
   │ 000000c0   400c  400d  400e  400f  4010  4011 │
   │ 00000120   4012  4013  4014  4015  4016  4017 │
   │ 00000180   4018  4019  401a  401b  401c  401d │
   │ 000001e0   401e  401f  4020  4021  4022  4023 │
   └────────────────────────────────────────────────┘
```

The display changes when you resize the window

Purpose	Displays the contents of memory
Editable?	Yes—you can edit the data (but not the addresses)
Modes	Auto (assembly display only), assembly, and mixed
Created	Automatically
Affected by	The MEM command

The MEMORY window has two parts:

❑ **Addresses.** The first column of numbers identifies the addresses of the first column of displayed data. No matter how many columns of data you display, only one address column is displayed. Each address in this column identifies the address of the data immediately to its right.

❑ **Data.** The remaining columns display the values at the listed addresses. You can display more data by making the window wider and/or longer.

The first MEMORY window above has one column of data, so each new address is incremented by 10_{16}. Although the second window shows six columns of data, there is still only one column of addresses; the first value is at address 0x0000 0000, the second at address 0x0000 0010, etc.; the seventh value (first value in the second row) is at address 0x0000 0060, the eighth at address 0x0000 0070, etc.

As you run programs, some memory values change as the result of program execution. The debugger highlights changed values. Depending on how you configure memory for your application, some locations may be invalid/unconfigured. The debugger also highlights these locations (by default, it shows these locations in red).

If you want to view different memory locations, use the MEM command to display a different block of memory. The basic syntax for this command is:

mem *address*

When you enter this command, the debugger changes the memory display so that *address* becomes the first displayed location (it's displayed in row 1, column 1).

CPU window

register name		
register contents		

The display changes when you resize the window

Purpose	Shows the contents of the '340 registers
Editable?	Yes—you can edit the value of any displayed register
Modes	Auto (assembly display only), assembly, and mixed
Created	Automatically
Affected by	Data-management commands

As you run programs, some values displayed in the CPU window change as the result of program execution. The debugger highlights changed values.

I/O window

```
         ┌─I/O ─────────────────────────────────
         │ VESYNC   0000      HESYNC    0000
 register│ VEBLNK   0000      HEBLNK    0000
 name    │ VSBLNK   0000      HSBLNK    0000
         │ VCOUNT   0000      HCOUNT    0000
         │ VTOTAL   0000      HTOTAL    0000
         │ DPYCTL   0000      DPYSTRT   0000
         │ DPYINT   0000      DPYTAP    0000
         │ DPYADR   0000      REFADR    0000
         │ CONTROL  0000      PSIZE     0000
         │ INTENB   0000      INTPEND   0000
         │ CONVSP   0000      CONVDP    0000
 register│ HESERR   0000      SCOUNT    0000
 contents│ CONVMP   0000      CONFIG    0000
         │ SETVCNT  0000      SETHCNT   0000
         │ HSTCTLL  0000      HSTCTLH   0000
         │ HSTDATA  0000      BSFLTST   0000
         │ HSTADR   00000000  BSFLTD    00000000
         │ PMASK    00000000  DINC      00000000
         │ DPYNX    00000000  DPYST     00000000
         │ DPYMSK   0000
```

Purpose   Shows the contents of the '340 I/O registers

Editable?   Yes—you can edit the value of any displayed register

Modes   Auto (assembly display only), assembly, and mixed

Created   With the IOREGS command

Affected by Data-management commands

As you run programs, some values displayed in this window may change as the result of program execution. The debugger highlights changed values.

FPU window

```
┌─FPU
 RA0   0.000000E+000      RB0    0.000000E+000
 RA1   0.000000E+000      RB1    0.000000E+000
 RA2   0.000000E+000      RB2    0.000000E+000
 RA3   0.000000E+000      RB3    0.000000E+000
 RA4   0.000000E+000      RB4    0.000000E+000
 RA5   0.000000E+000      RB5    0.000000E+000
 RA6   0.000000E+000      RB6    0.000000E+000
 RA7   0.000000E+000      RB7    0.000000E+000
 RA8   0.000000E+000      RB8    0.000000E+000
 RA9   0.000000E+000      RB9    0.000000E+000
 CSTATIS   00001000       CCPMFOG   ffe00420
```

register name

register contents

Purpose	Shows the contents of the '34082 registers
Editable?	Yes—you can edit the value of any displayed register
Modes	Auto (assembly display only), assembly, and mixed
Created	With the FPUREGS command
Affected by	Data-management commands

The FPU window shows the '34082 A- and B-file registers in double-precision scientific notation. It also shows the '34082 status and configuration registers as hexadecimal values.

As you run programs, some values displayed in this window may change as the result of program execution. The debugger highlights changed values.

Note:

The FPU window can be displayed only if you've invoked the debugger with the –mc option.

DISP windows

```
                  ┌DISP: str────┐                    ┌DISP: str.f4──────┐
                  │ a    84     │                    │ [0] 44276127     │
                  │ b    86     │                    │ [1] 1778712578   │
  structure ───── │ c    172    │                    │ [2] 555492660    │
  members         │ f1   1      │                    │ [3] 356713217    │
                  │ f2   7      │                    │ [4] 138412802    │
                  │ f3   0x18740001                  │ [5] 182452229    │
  member ──────── │ f4   [...]  │                    │ [6] 35659888     │
  values          └─────────────┘                    │ [7] 37749506     │
                                                     │ [8] 134742016    │
                       *This member is an array, and* │ [9] 138412801    │
                      *you can display its contents in*└──────────────────┘
                            *a second DISP window*
```
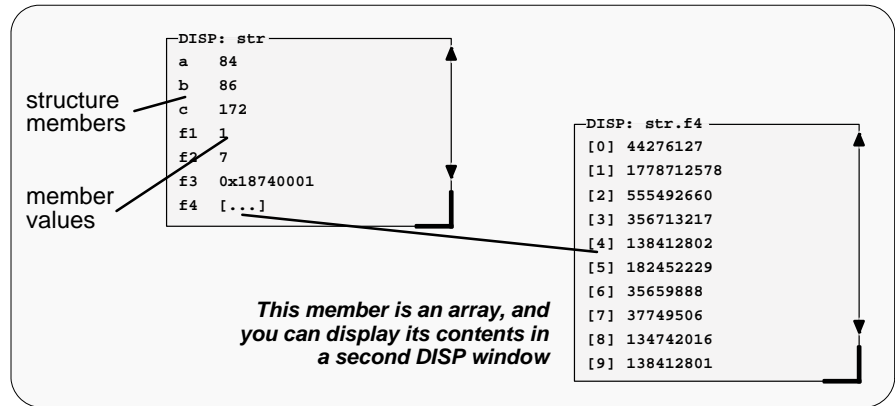
Purpose	Displays the members of a selected structure, array or pointer, and the value of each member
Editable?	Yes—you can edit individual values
Modes	Auto (C display only) and mixed
Created	With the DISP command
Affected by	DISP command

A DISP window is similar to a WATCH window, but it shows the values of an entire array or structure instead of a single value. Use the DISP command to open a DISP window; the basic syntax is:

disp *expression*

Data is displayed in its natural format: :

❏ Integer values are displayed as decimal numbers.
❏ Floating-point values are displayed in floating-point format.
❏ Pointers are displayed as hexadecimal addresses (with a 0x prefix).
❏ Enumerated types are displayed symbolically.

If any of the displayed members are arrays, structures, or pointers, you can bring up additional DISP windows to display their contents—up to 120 DISP windows can be open at once.

WATCH window

```
                        ┌─WATCH ─────────────────┐
                        │1:  A0    -1            ▲│
  watch index ──────────│2:  X+X   4             │
                        │3:  PC    0x00400064    ▼│
                        └────────────────────────┘
                              label    current value
```

Purpose	Displays the values of selected expressions
Editable?	Yes—you can edit the value of any expression whose value specifies a storage location (in registers or memory). In the window above, for example, you could edit the value of PC but couldn't edit the value of X+X.
Modes	Auto, assembly, and mixed
Created	With the WA command
Affected by	WA, WD, and WR commands

The WATCH window helps you to track the values of arbitrary expressions, variables, and registers. Use the WA command for this; the syntax is:

wa *expression* [, *label*]

WA adds *expression* to the WATCH window. (If there's no WATCH window, then WA also opens a WATCH window).

To delete individual entries from the WATCH window, use the WD command. To delete all entries at once and close the WATCH window, use the WR command.

Although the CPU window displays register contents, you may not be interested in the values of all these registers. In this situation, it is convenient to use the WATCH window to track the values of the specific registers you're interested in.

6.3 Cursors

The debugger display has three types of cursors:

❑ The **command-line cursor** is a block-shaped cursor that identifies the current character position on the command line. Arrow keys *do not affect* the position of this cursor.

```
┌─COMMAND──────────────────────────────────────────┐
│ file sample.c                                   ▲
│ go meminit                                      │
│ go call                                         │
│ go c_int00                                      │
│ restart                                         ▼
│ >>> go main █                                    │
└─────────────────────────────────────────────────┘
```

command line cursor
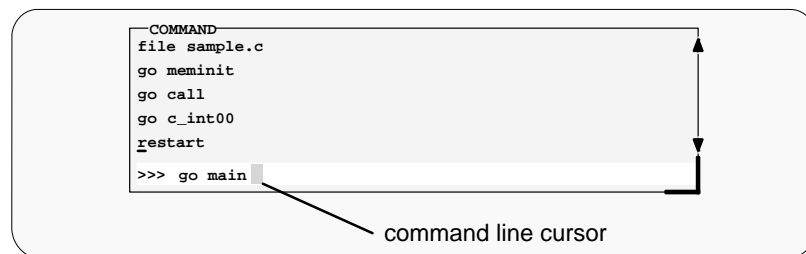
❑ The **mouse cursor** is a block-shaped cursor that tracks mouse movements over the entire display. This cursor is controlled by the mouse driver installed on your system; if you haven't installed a mouse, you won't see a mouse cursor on the debugger display.

❑ The **current-field cursor** identifies the current field in the active window. This is the hardware cursor that is associated with your EGA or VGA card. Arrow keys *do* affect this cursor's movement.

```
┌─CPU────────────────────────────────────────────┐
│ B0  00f0007    B1  0000075    B2  0000000      ▲
│ 6              5              3                 │
│ B3  0000000    B4  0000000    B5  0000000       │
│ 5  _           7              0                 ▼
│ B6  0000000    B7  0000000    B8  0000000       │
│ 0              0              0                 │
│ B9  0000000    B10 0000010    B11 0000000       │
│ 0              2              0                 │
└────────────────────────────────────────────────┘
```

current field cursor
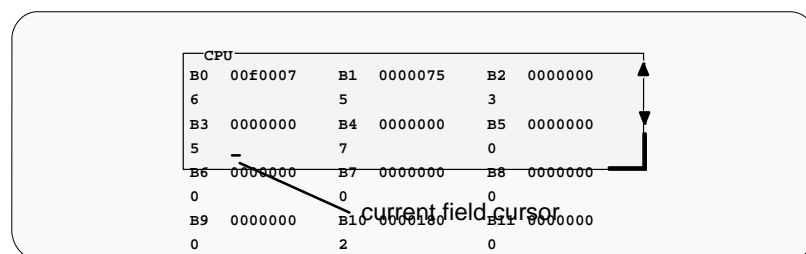
6.4 The Active Window

The windows in the debugger display aren't fixed in their position or in their size. You can resize them, move them around, and, in some cases, close them. The window that you're going to move, resize, or close must be **active**.

You can move, resize, or close *only one window at a time*; thus, only one window at a time can be the **active window**. Whether or not a window is active doesn't affect the debugger's ability to update information in a window — it affects only your ability to manipulate a window.

Identifying the active window

The debugger highlights the active window. When windows overlap on your display, the debugger pops the active window to be on top of other windows.

You can alter the active window's border style and colors if you wish; Figure 6–4 illustrates the default appearance of an active window and an inactive window.

Figure 6–4. Default Appearance of an Active and an Inactive Window

An active window (default appearance)

```
┌─COMMAND──────────────────┐
│ TMS340 Debugger Version 5.00          ▲
│ Copyright (c) 1990, Texas Instrume   │
│
│ Loading sample.out
│ go main                               ▼
│ >>>
```

This window is highlighted to show that it is active

An inactive window (default appearance)

```
┌─COMMAND─────
│ TMS340 Debugger Version 5.00                              ▲
│ Copyright (c) 1990, Texas Instruments Incorporated
│
│ Loading sample.out
│ go main                                                   ▼
│ >>>
```

This window is not highlighted and is not active

Note: On **monochrome monitors**, the border and selection corner are highlighted as shown in the illustration. On **color monitors**, the border and selection corner are highlighted as shown in the illustration, but they also change color (by default, they change from white to yellow).

Selecting the active window

You can use one of several methods for selecting the active window:

↖ 1) Point to any location within the boundaries or on any border of the desired window.

2) Click the left mouse button.

Note that if you point within the window, you might also select the current field. For example:

❑ If you point inside the CPU window, then the register you're pointing at becomes active, and the debugger treats any text that you type as a new register value. If you point inside the MEMORY window, then the address value you're pointing at becomes active, and the debugger treats any text that you type as a new memory value.

Press ESC *to get out of this.*

❑ If you point inside the DISASSEMBLY or FILE window, you'll set a breakpoint on the statement you're pointing to.

Press the button again to clear the breakpoint.

F6 This key hops through the windows on your display, making each one active in turn and making the previously active window inactive. Pressing this key highlights one of the windows, showing you that the window is active. Pressing F6 again makes a different window active. Press F6 as many times as necessary until the desired window becomes the active window.

win   The WIN command allows you to select the active window by name. The format of this command is:

win   *WINDOW NAME*

Note that the *WINDOW NAME* is in uppercase (matching the name exactly as displayed). You can spell out the entire window name, but you really need specify only enough letters to identify the window.

For example, to select the DISASSEMBLY window as the active window, you could enter either of these two commands:

      win DISASSEMBLY⏎

or   **win DISA** ⏎

If several windows of the same type are visible on the screen, don't use the WIN command to select one of them.

If you supply an ambiguous name (such as C, which could stand for CPU or CALLS), the debugger selects the first window it finds whose name matches the name you supplied. If the debugger doesn't find the window you asked for (because you closed the window or misspelled the name), then the WIN command has no effect.

6.5 Manipulating Windows

Windows don't have a fixed size or position — you can change their sizes and you can move them around. This section tells you how to do this.

Note:

You can resize or move any window, but first the window must be **active**. For information about selecting the active window, refer to Section 6.4 (page xx).

Resizing a window

The minimum window size is three lines by four characters. The maximum window size varies, depending on which screen size you're using, but you can't make a window larger than the screen.

There are two basic ways to resize a window:

❑ You can resize a window by using the mouse.
❑ You can resize a window by using the SIZE command.

1) Point to the lower right corner of the window. This corner is highlighted —here's what it looks like:

```
┌COMMAND──────────────────┐
│TMS340 Debugger Version 5.00 │
│Copyright (c) 1990, Texas Instr │
│Loading sample.out          │
│                            │
│go main                     │
│>>>█                        │
└─────────────────────────┘
```
lower right corner
(highlighted)
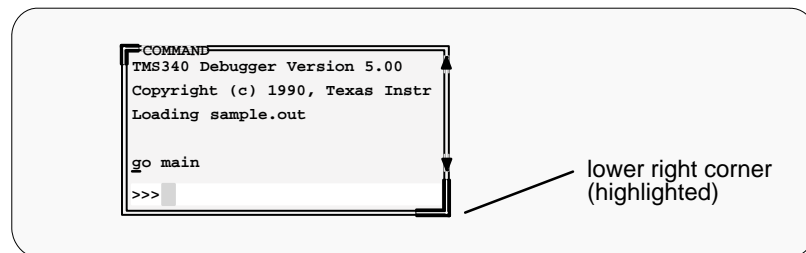
2) Grab the highlighted corner by pressing one of the mouse buttons; while pressing the button, move the mouse in any direction. This resizes the window.

3) Release the mouse button when the window reaches the desired size.

size   The SIZE command allows you to size the active window. The format of this command is:

size   [*width*, *length*]

You can use the SIZE command in one of two ways:

Method 1   Supply a specific *width* and *length*

Method 2   Omit the *width* and *length* parameters and use arrow keys to interactively resize the window.

SIZE, method 1: Use *width* and *length* parameters. Valid values for the width and length depend on the screen size and the window position on the screen. Table 6–1 lists the minimum and maximum window sizes.

Table 6–1. Width and Length Limits for Window Sizes

Screen size	Debugger option	Valid widths	Valid lengths
80 characters by 25 lines	none	4 through 80	3 through 24
80 characters by 43 lines (EGA)	–b	4 through 80	3 through 42
80 characters by 50 lines (VGA)			3 through 49
120 characters by 43 lines	–bb	4 through 120	3 through 42
132 characters by 43 lines	–bbb	4 through 132	3 through 42
80 characters by 60 lines	–bbbb	4 through 80	3 through 59
100 characters by 60 lines	–bbbbb	4 through 100	3 through 59

Note: To use a larger screen size, you must invoke the debugger with one of the –b options.

The maximum sizes assume that the window is in the upper left corner (beneath the menu bar). If a window is in the middle of the display, for example, you can't size it to the maximum height and width—you can size it only to the right and bottom screen borders. The easiest way to make a window as large as possible is to zoom it, as described on page xxv.

If you want to use commands to make the CALLS window 8 characters wide by 20 lines long, you could enter:

```
win  CALLS  ⏎
size 8, 20  ⏎
```

SIZE, method 2: Use arrow keys to interactively resize the window. If you enter the SIZE command without *width* and *length* parameters, you can use arrow keys to size the window:

⬇   Makes the active window one line longer.

⬆   Makes the active window one line shorter.

⬅   Makes the active window one character narrower.

➡   Makes the active window one character wider.

> When you're finished using the cursor keys, you *must* press `ESC` or `⏎` .

For example, if you want to make the CPU window three lines longer and two characters narrower, you can enter:

```
win  CPU  ⏎
size  ⏎
⬇ ⬇ ⬇        ⬅ ⬅          ESC
```

Zooming a window

Another way to resize the active window is to zoom it. Zooming a window makes it as large as possible, so that it takes up the entire display (except for the menu bar) and hides all the other windows. Unlike the SIZE command, zooming is not affected by the window's position in the display.

To "unzoom" a window, repeat the same steps you used to zoom it. This will return the window to its prezoom size and position.

There are two basic ways to zoom a window:

❑ You can zoom a window by using the mouse.

❑ You can zoom a window by using the ZOOM command.

1) Point to the upper left corner of the window. This corner is highlighted—here's what it looks like:

```
                                    ┌COMMAND────────────────┐
                                    │TMS340 Debugger Version 5.00 │▲
   upper left corner                │Copyright (c) 1990, Texas Instr │
   (highlighted)                    │Loading sample.out          │
                                    │                            │
                                    │go main                     │▼
                                    │>>>                         │
                                    └────────────────────────────┘
```
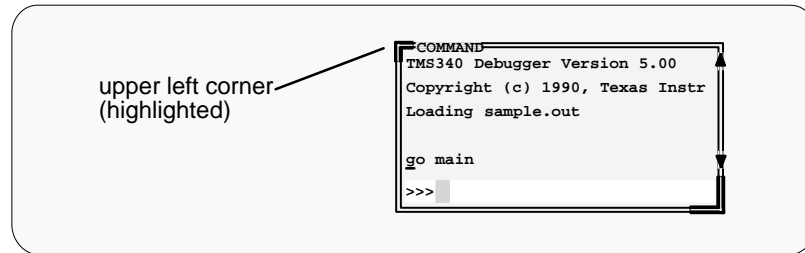
2) Click the left mouse button.

zoom You can also use the ZOOM command to zoom/unzoom the window. The format for this command is:

zoom

Moving a window

The windows in the debugger display don't have fixed positions—you can move them around.

There are two ways to move a window:

❑ You can move a window by using the mouse.
❑ You can move a window by using the MOVE command.

1) Point to the left or top edge of the window.

```
                                    ┌COMMAND────────────────┐
   Point to the top edge            │TMS340 Debugger Version 5.00 │▲
   or the left edge                 │Copyright (c) 1990, Texas Instr │
                                    │Loading sample.out          │
                                    │                            │
                                    │go main                     │▼
                                    │>>>                         │
                                    └────────────────────────────┘
```
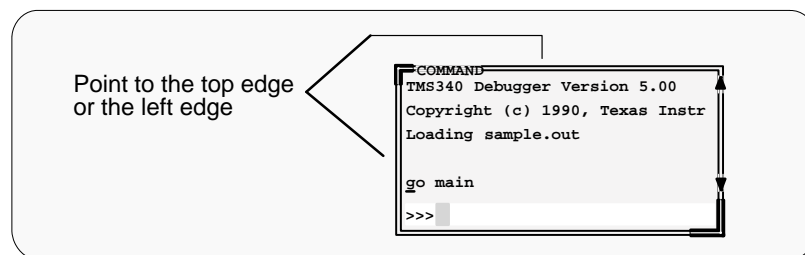
⬛️ 2) Grab the window by pressing the left mouse button, but don't release the button; now move the mouse in any direction.

☐☐ 3) Release the mouse button when the window is in the desired position.

move The MOVE command allows you to move the active window. The format of this command is:

move [*X position*, *Y position* [, *width*, *length*]]

You can use the MOVE command in one of two ways:

Method 1 Supply a specific *X position* and *Y position*

Method 2 Omit the *X position* and *Y position* parameters and use arrow keys to interactively resize the window

MOVE, method 1: Use the *X position* and *Y position* parameters. You can move a window by defining a new XY position for the window's upper left corner. Valid X and Y positions depend on the screen size and the window size. Table 6–2 lists the minimum and maximum XY positions.

Table 6–2. Minimum and Maximum Limits for Window Positions

Screen size	Debugger option	Valid X positions	Valid Y positions
80 characters by 25 lines	none	0 through 76	1 through 22
80 characters by 43 lines (EGA) 80 characters by 50 lines (VGA)	–b	0 through 76	1 through 40 1 through 47
120 characters by 43 lines	–bb	0 through 116	1 through 40
132 characters by 43 lines	–bbb	0 through 128	1 through 40
80 characters by 60 lines	–bbbb	0 through 76	1 through 57
100 characters by 60 lines	–bbbbb	0 through 106	1 through 57

Note: To use a larger screen size, you must invoke the debugger with one of the –b options.

The maximum values assume that the window is as small as possible; for example, if a window is half as tall as the screen, you won't be able to move its upper left corner to an X position on the bottom half of the screen.

If you want to use commands to move the DISASSEMBLY position to a place in the upper left area of the display, you might enter:

```
win DISASSEMBLY  ⮐
move 5, 6  ⮐
```

MOVE, method 2: Use arrow keys to interactively move the window. If you enter the MOVE command without *X position* and *Y position* parameters, you can use arrow keys to move the window:

⬇   Moves the active window down one line.
⬆   Moves the active window up one line.
⬅   Moves the active window left one character position.
➡   Moves the active window right one character position.

> When you're finished using the cursor keys, you *must* press ⌈ESC⌉ or ⌈⬐⌉ .

For example, if you want to move the COMMAND window up two lines and right five characters, you can enter:

```
win  COM ⬐
move ⬐
⬆ ⬆      ➡ ➡ ➡ ➡ ➡        ESC
```

Note:

If you choose, you can resize a window at the same time you move it. To do this, use the *width* and *length* parameters in the same way that they are used for the SIZE command.

6.6   Manipulating a Window's Contents

Although you may be concerned with changing the way windows appear in the display — where they are and how big/small they are — you'll usually be interested in something much more important: *what's in the windows*. Some windows contain more information than can be displayed on a screen; others contain information that you'd like to change. This section tells you how to view the hidden portions of data within a window and which data can be edited.

Note:

You can scroll and edit only the **active window**. For information about selecting the active window, refer to Section 6.4 (page xx).

Scrolling through a window's contents

If you resize a window to make it smaller, you may hide information. Sometimes, a window may contain more information than can be displayed on a screen. In these cases, the debugger allows you to scroll information up and down within the window.

There are two ways to view hidden portions of a window's contents:

❏   You can use the mouse to scroll the contents of the window.
❏   You can use function keys and arrow keys.

You can use the mouse to point to the scroll arrows on the righthand side of the active window. This is what the scroll arrows look like:

```
┌─FILE: sample.c─────────────────┐
00038 extern call();
00039 extern meminit();
00040 main()
00041 {
00042      register int i = 0;
00043      int j = 0, k = 0;
00044
00045      meminit();
00046      for (i = 0, i , 0x50000; i++)
00047      {
00048           call(i);
00049           if (i & 1) j += i;
00050           aai[k][k] = j;
00051           if (!(i & 0xFFFF)) k++;
00052      }
```

scroll up

scroll down
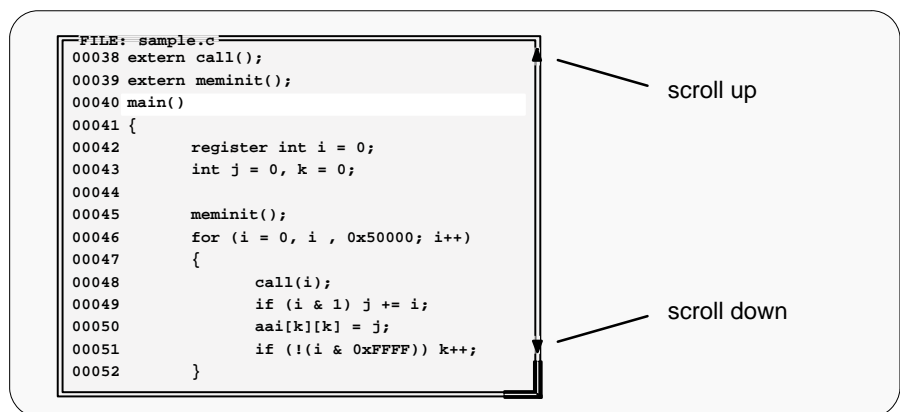
To scroll window contents up or down:

1) Point to the appropriate scroll arrow.

2) Press the left mouse button; continue to press it until the information you're interested in is displayed within the window.

3) Release the mouse button when you're finished scrolling.

You can scroll up/down one line at a time by pressing the mouse button and releasing it immediately.

[key]

In addition to scrolling, the debugger supports the following methods for moving through a window's contents.

(PAGE UP)

The page-up key scrolls up through the window contents, one window length at a time. You can use (CONTROL) (PAGE UP) to scroll up through an array of structures displayed in a DISP window.

(PAGE DOWN)

The page-down key scrolls down through the window contents, one window length at a time. You can use (CONTROL) (PAGE DOWN) to scroll down through an array of structures displayed in a DISP window.

(HOME)   When the FILE window is active, pressing (HOME) adjusts the window's contents so that the first line of the text file is at the top of the window. You can't use (HOME) outside of the FILE window.

(END)   When the FILE window is active, pressing (END) adjusts the window's contents so that the last line of the file is at the bottom of the window. You can't use (END) outside of the FILE window.

(↑)   Moves the field cursor up one line at a time.

(↓)   Moves the field cursor down one line at a time.

(←)   In the FILE window, scrolls the display left eight characters at a time. In other windows, moves the field cursor left one field; at the first field on a line, wraps back to the last fully displayed field on the previous line.

(→)   In the FILE window, scrolls the display right eight characters at a time. In other windows, moves the field cursor right one field; at the last field on a line, wraps around to the first field on the next line.

Editing the data displayed in windows

You can edit the data displayed in the MEMORY, CPU, DISP, and WATCH windows by using an overwrite "click and type" method or by using commands

that change the values. (This is described in detail in Section 11.3, *Basic Methods for Changing Data Values*, page 11-4.)

Note:

In these windows, the "click and type" method of selecting data for editing —pointing at a line and pressing [F9] or clicking a mouse button — does not allow you to modify data.

In the FILE and DISASSEMBLY windows, pressing [F9] or a mouse button sets or clears a breakpoint on any line of code that you select. You can't modify text in a FILE or DISASSEMBLY window.

In the CALLS window, pressing [F9] or a mouse button shows the source for the function named on the selected line.

6.7 Closing a Window

The debugger opens various windows on the display according to the debugging mode you select. When you switch modes, the debugger may close some windows and open others. Additionally, you may choose to open DISP and WATCH windows.

Most of the windows remain open — you can't close them. However, you can close the CALLS, DISP, and WATCH windows.

❑ To close the CALLS window:

    1)   Make the CALLS window the active window.

    2)   Press ⌑F4⌑.

❑ To close a DISP window:

    1)   Make the appropriate DISP window the active window.

    2)   Press ⌑F4⌑.

If the DISP window that you close has any children, they are closed also.

❑ To close the WATCH window, enter:

wr ⏎

Entering and Using Commands

The debugger provides you with several methods for entering commands and accomplishing other tasks within the debugger environment. There are several ways to enter commands: from the command line, from pulldown menus, with a mouse, and with function keys. Mouse and function key use differs from situation to situation, and is described throughout this book whenever applicable. Certain specific rules apply to entering commands and using pulldown menus, however, and this chapter includes this information.

	Synopsis Page	Topic	

7.1 Entering Commands From the Command Line

The debugger supports a complete set of commands that help you to control and monitor program execution, customize the display, and perform other tasks. These commands are discussed in the various sections throughout this book, as they apply to the current topic. Chapter 14 summarizes all of the debugger commands with an alphabetic reference.

Although there are a variety of methods for entering most of the commands, *all* of the commands can be entered by typing them on the command line in the COMMAND window. Figure 7–1 shows the COMMAND window.

Figure 7–1. The COMMAND Window
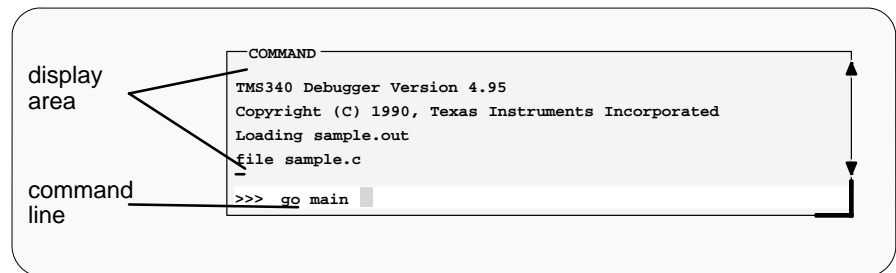
The COMMAND window serves two purposes:

❑ The **command line** portion of the window provides you with an area for entering commands. For example, the command line in Figure 7–1 shows that a GO command was typed in (but not yet entered).

❑ The **display area** provides the debugger with an area for echoing commands, displaying command output, or displaying errors and messages for you to read. For example, the command output in Figure 7–1 shows the messages that are displayed when you first bring up the debugger and also shows that a FILE command was entered.

If you enter a command by using an alternate method (using the mouse, a pulldown menu, or function keys), the COMMAND window doesn't echo the entered command.

How to type in and enter commands

You can type a command at almost any time; the debugger automatically places the text on the command line when you type. When you want to enter a command, just type—no matter which window is active. You don't have to worry about making the COMMAND window active or moving the field cursor to the command line. When you start to type, the debugger usually assumes that you're typing a command and puts the text on the command line (except under certain circumstances, which are explained on the next page). Commands themselves are not case sensitive, although some parameters (such as window names) are.

To execute a command that you've typed, just press ⏎. The debugger then:

1) Echoes the command to the display area,
2) Executes the command and displays any resulting output, and
3) Clears the command line when command execution completes.

Once you've typed a command, you can edit the text on the command line with these keystrokes:

To...	Press...
Move back over text without erasing characters	`CTRL` `H` **or** `BACK SPACE`
Move forward through text without erasing characters	`CTRL` `L`
Move back over text while erasing characters	`DELETE`
Move forward through text while erasing characters	`SPACE`
Insert text into the characters that are already on the command line	`INSERT`

Note:

❏ You cannot use the arrow keys to move through or edit text on the command line.

❏ Typing a command doesn't make the COMMAND window the active window.

❏ If you press ⏎ when the cursor is in the middle of text, the debugger truncates the input text at the point where you press ⏎.

Sometimes, you can't type a command

At most times, you can press any alphanumeric or punctuation key on your keyboard (any printable character); the debugger interprets this as part of a command and displays the character on the command line. In a few instances, however, pressing an alphanumeric key is not interpreted as information for the command line.

❑ When you're pressing the ⌷ALT⌷ key, typing certain letters causes the debugger to display a pulldown menu.

❑ When a pulldown menu is displayed, typing a letter causes the debugger to execute a selection from the menu.

❑ When you're pressing the ⌷CONTROL⌷ key, pressing ⌷H⌷ or ⌷L⌷ moves the command-line cursor backward or forward through the text on the command line.

❑ When you're editing a field, typing enters a new value in the field.

❑ When you're using the MOVE or SIZE command interactively, pressing keys affects the size or position of the active window. Before you can enter any more commands, you must press ⌷ESC⌷ to terminate the interactive moving or sizing.

❑ When you've brought up a dialog box, typing enters a parameter value for the current field in the box.

Using the command history

The debugger keeps an internal list, or **command history**, of the commands that you enter. It remembers the last 100 commands that you entered. If you want to re-enter a command, you can move through this list, select a command that you've already executed, and re-execute it.

Use these keystrokes to move through the command history.

To...	Press...
Repeat the last command that you entered	⌷F2⌷
Move forward through the list of executed commands, one by one	⌷SHIFT⌷ ⌷TAB⌷
Move backward through the list of executed commands, one by one	⌷TAB⌷

As you move through the command history, the debugger displays the commands, one by one, on the command line. When you see a command that you want to execute, simply press ⏎ to execute the command. You can also edit these displayed commands in the same manner that you can edit new commands.

Clearing the display area

Occasionally, you may want to completely blank out the display area of the COMMAND window; the debugger provides a command for this:

cls   Use the CLS command to clear all displayed information from the display area. The format for this command is:

cls

7.2 Using the Menu Bar and the Pulldown Menus

In all three of the debugger displays, you'll see a menu bar at the top of the screen. The menu selections offer you an alternative method for entering many of the debugger commands. Figure 7–2 points out the menu bar in a mixed-mode display. There are several ways to use the selections on the menu bar, depending on whether the selection has a pulldown menu or not.

Figure 7–2. The Menu Bar in the Debugger Display
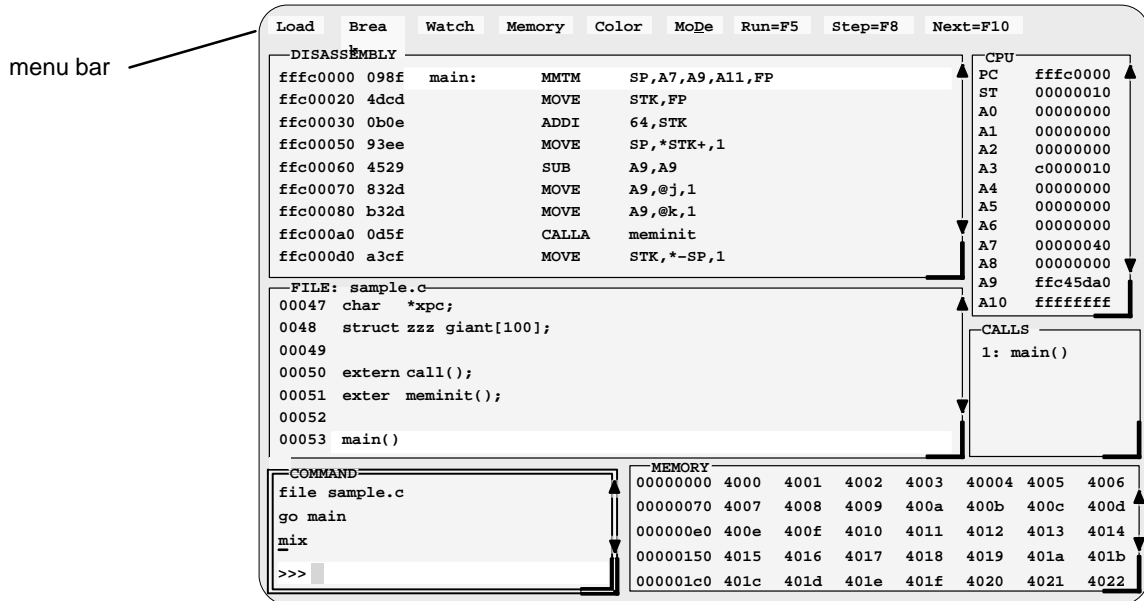
menu bar

Several of the selections on the menu bar have pulldown menus; if they could all be pulled down at once, they'd look like Figure 7–3.

Figure 7–3. All of the Pulldown Menus
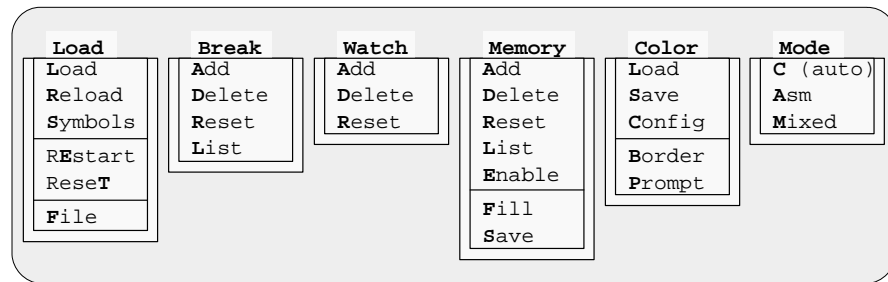
Note that the menu bar and associated pulldown menus occupy fixed positions on the display. Unlike windows, you can't move, resize, or cover the menu bar or pulldown menus.

Using the pulldown menus

There are several ways to display the pulldown menus and then execute your selections from them. Executing a command from a menu is similar to executing a command by typing it in.

❑ If you select a command that has no parameters, then the debugger executes the command as soon as you select it.

❑ If you select a command that has one or more parameters, the debugger displays a **dialog box** when you make your selection. A dialog box offers you the chance to type in the parameter values for the command.

The following paragraphs describe several methods for selecting commands from the pulldown menus.

Mouse method 1

1) Point the mouse cursor at one of the appropriate selections in the menu bar.

2) Press the left mouse button, but don't release the button.

3) While pressing the mouse button, move the mouse downward until your selection is highlighted on the menu.

4) When your selection is highlighted, release the mouse button.

Mouse method 2

1) Point the cursor at one of the appropriate selections in the menu bar.

2) Click the left mouse button. This displays the menu until you are ready to make a selection.

3) Point the mouse cursor at your selection on the pulldown menu.

4) When your selection is highlighted, click the left mouse button.

key

Keyboard method 1

ALT 1) Press the ALT key; don't release it.

X 2) Press the key that corresponds to the highlighted letter in the selection name; release both keys. This displays the menu and freezes it.

X 3) Press and release the key that corresponds to the highlighted letter of your selection in the menu.

Keyboard method 2

ALT 1) Press the ALT key; don't release it.

X 2) Press the key that corresponds to the highlighted letter in the selection name; release both keys. This displays the menu and freezes it.

↓ ↑ 3) Use the arrow keys to move up and down through the menu.

↵ 4) When your selection is highlighted, press ↵.

Escaping from the pulldown menus

❏ If you display a menu and then decide that you don't want to make a selection from this menu, you can:

■ Press ESC.

or

■ Point the mouse outside of the menu; press and then release the left mouse button.

❏ If you pull down a menu and see that it is not the menu you wanted, you can point the mouse at another entry and press the left mouse button, or you can use the ← and → keys to display adjacent menus.

Entering parameters in a dialog box

Many of the debugger commands have parameters. When you execute these commands from menus, you must have some way of providing parameter values. The debugger allows you to do this by displaying a **dialog box** that asks for these values.

Entering parameter values in a dialog box is much like entering commands on the command line:
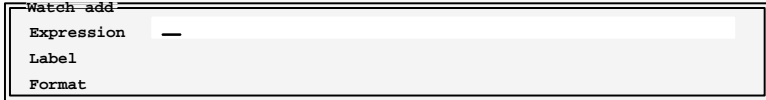
❑ If you press ⌅ in the middle of a string of text, the debugger truncates the string at that point.

❑ When you display a dialog box for the first time during a debugging session, the parameter fields are empty. When you bring up the same dialog box again, though, the box displays the last values that you entered. (This is similar to having a command history.) If you want to use the same value, just press ⌅.

❑ You can edit what you type (or values that remain from previous entry) in the same way that you can edit text on the command line.

When the dialog box displays more than one parameter, you can use the arrow keys to move from parameter to parameter. You can omit entries for optional parameters, but the debugger won't allow you to skip required parameters.
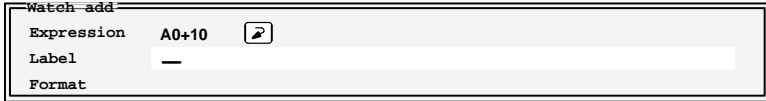
For example, the Add entry on the Watch menu is equivalent to the WA command. This command has three parameters:

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

When you select Add from the menu, the debugger displays a dialog box that asks you for this parameter information. The dialog box looks like this:

```
┌─Watch add──────────────────────────────────────────┐
│  Expression    ▬                                     │
│  Label                                               │
│  Format                                              │
└─────────────────────────────────────────────────────┘
```

You can enter an *expression* just as you would if you were typing the WA command, and then press ⌅. The cursor moves down to the next parameter:

```
┌─Watch add──────────────────────────────────────────┐
│  Expression    A0+10    ⌅                            │
│  Label         ▬                                     │
│  Format                                              │
└─────────────────────────────────────────────────────┘
```

In this case, the next two parameters *(label* and *format)* are optional. If you want to enter a parameter, you may do so; if you don't want to use these parameters, don't type anything in their fields—just press⌅ . When you've entered ⌅ for the final parameter, the debugger closes the dialog box and executes the command with the parameter values you supplied.

Using menu bar selections that don't have pulldown menus

These three menu bar selections are single-level entries without pulldown menus:

> **Run=F5   Step=F8   Next=F10**

There are two ways to execute these choices.

↖   1) Point the cursor at one of these selections in the menu bar.

   2) Click the left mouse button.

This executes your choice in the same manner as typing in the associated command without its optional *expression* parameter.

F5   Pressing this key is equivalent to typing in the RUN command without an *expression* parameter.

F8   Pressing this key is equivalent to typing in the STEP command without an *expression* parameter.

F10   Pressing this key is equivalent to typing in the NEXT command without an *expression* parameter.

How the menu selections correspond to commands

The following sample screens illustrate the relationship of the debugger commands to the menu bar and pulldown menus.

program execution commands

> **Run=F5** — RUN command (without a parameter)
>
> **Step=F8** — STEP command (without a parameter)
>
> **Next=F10** — NEXT command (without a parameter)

breakpoint commands

> **Break**
> **A**dd ———————— BA command
> **D**elete ———————— BD command
> **R**eset ———————— BR command
> **L**ist ———————— BL command

file/load commands

```
     Load
   Load                 ——————————  LOAD command
   Reload               ——————————  RELOAD command
   Symbols              ——————————  SLOAD command
   REstart              ——————————  RESTART command
   ReseT                ——————————  RESET command
   File                 ——————————  FILE command
```

breakpoint commands

```
     Break
   Add                  ——————————  BA command
   Delete               ——————————  BD command
   Reset                ——————————  BR command
   List                 ——————————  BL command
```

watch commands

```
     Watch
   Add                  ——————————  WA command
   Delete               ——————————  WD command
   Reset                ——————————  WR command
```

memory commands

```
     Memory
   Add                  ——————————  MA command
   Delete               ——————————  MD command
   Reset                ——————————  MR command
   List                 ——————————  ML command
   Enable               ——————————  MAP command
   Fill                 ——————————  FILL command
   Save                 ——————————  MS command
```

screen-configuration commands

```
     Color
   Load                 ——————————  SCONFIG command
   Save                 ——————————  SSAVE command
   Config               ——————————  SCOLOR command
   Border               ——————————  BORDER command
   Prompt               ——————————  PROMPT command
```

mode commands

```
     Mode
   C (auto)             ——————————  C command
   Asm                  ——————————  ASM command
   Mixed                ——————————  MIX command
```
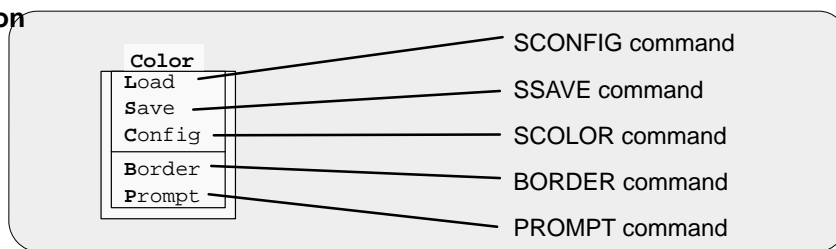
7.3 Entering Commands From a Batch File

You can place debugger commands in a batch file and execute the file from within the debugger environment. This is useful, for example, for setting up a memory map that contains several MA commands followed by a MAP command that enables memory mapping.

---

take Use the TAKE command to tell the debugger to read and execute commands from a batch file. A batch file can call another batch file; they can be nested in this manner up to 10 deep. To halt the debugger's execution of a batch file, press ESC .

The format for the TAKE command is:

take *batch filename* [*, suppress echo flag*]

❏ The *batch filename* parameter identifies the file that contains commands.

■ If you don't supply an extension, .cmd is used.

■ If you supply path information with the *filename*, the debugger looks for the file only in the specified directory.

■ If you don't supply path information with the *filename*, the debugger looks for the file in the current directory.

■ If the debugger can't find the file in the current directory, it looks in any directories that you identified with the D_DIR environment variable. You can set D_DIR within the DOS environment; the command for doing this is

SET D_DIR=*C:\pathname;C:\pathname*

This allows you to name several directories that the debugger can search. Remember that if you use D_DIR, you must set it *before you invoke the debugger*— the debugger doesn't recognize the DOS SET command. If you often use the same directories, it may be convenient to set D_DIR in your autoexec.bat file.

❑ By default, the debugger echoes the commands in the COMMAND window display area and updates the display as it reads commands from the batch file.

   ■ If you don't use the *suppress echo flag* parameter, or if you use it but supply a nonzero value, then the debugger behaves in the default manner.

   ■ If you would like to suppress the echoing and updating, use the value 0 for the *suppress echo flag* parameter.

7.4 Defining Your Own Command Strings

The debugger provides a shorthand method of entering often-used commands or command sequences. This processing is called *aliasing*. Aliasing enables you to define an alias name for the command(s) and then enter the alias name as if it were a debugger command.

To do this, use the ALIAS command. The syntax for this command is:

alias   [*alias name* [, "*command string*"]]

The primary purpose of the ALIAS command is to associate the *alias name* with the debugger command you've supplied as the *command string*. However, the ALIAS command is versatile and can be used in several ways:

❑ **Aliasing several commands.** The *command string* can contain more than one debugger command — just separate the commands with semicolons.

For example, suppose you always began a debugging session by loading the same object file, displaying the same C source file, and running to a certain point in the code. You could define an alias to do all these tasks at once:

```
alias init,"load test.out;file source.c;go main"
```

Now you could enter init instead of the three commands listed within the quote marks.

❑ **Supplying parameters to the command string.** The *command string* can define parameters that you'll supply later. To do this, use a percent sign and a number (%1) to represent the parameter that will be filled in later. The numbers should be consecutive (%1, %2, %3) unless you plan to reuse the same parameter value for multiple commands.

For example, suppose that every time you filled an area of memory you also wanted to display that block in the MEMORY window:

```
alias mfil,"fill %1, %2, %3;mem %1"
```

Then you could enter:

```
mfil 0xff8000000,0x18,0x11112222
```

The first value (0xFF800 0000) would be substituted for the first FILL parameter and the MEM parameter (%1). The second and third values would be substituted for the second and third FILL parameters (%2 and %3).

❑ **Listing all aliases.** To display a list of all the defined aliases, enter the ALIAS command with no parameters. The debugger will list the aliases and their definitions in the COMMAND window.

For example, assume that the init and mfil aliases had been defined as shown in the previous two examples. If you entered:

alias ⏎

you'd see:

```
   Alias       Command
   ------------------------------------------
   INIT    -->  load test.out;file source.c;go
   main
   MFIL    -->  fill %1,%2,%3;mem %1
```

❑ **Finding the definition of an alias.** If you know an alias name but are not sure of its current definition, enter the ALIAS command with just an alias name. The debugger will display the definition in the COMMAND window.

For example, if you had defined the init alias as shown in the first example above, you could enter:

alias init ⏎

Then you'd see:

```
"INIT" aliased as "load test.out; file source.c;go
main"
```

❑ **Nesting alias definitions.** You can include a defined alias name in the *command string* of another alias definition. This is especially useful when the command string would be longer than the debugger command line.

Notes:

Individual commands within a command string are limited to an expanded length of 132 characters. The expanded length of the command includes the length of any substituted parameter values.

To redefine an alias, re-enter the ALIAS command with the same alias name and a new command string. To get rid of an alias, use the UNALIAS command:

unalias *alias name*

Alias definitions are lost when you exit the debugger. If you want to reuse aliases, define them in a batch file.

7.5   Entering Operating-System Commands

The debugger provides a simple method for entering operating-system commands without explicitly exiting the debugger environment. To do this, use the SYSTEM command. The format for this command is:

system   ["*operating-system command*" [, *flag*]]

The SYSTEM command behaves in one of two ways, depending on whether or not you supply an operating-system command as a parameter:

❑   If you enter the command with an operating-system parameter, then you stay within the debugger environment.

❑   If you enter the command without parameters, the debugger opens a *system shell*. This means that the debugger will blank the debugger display and temporarily exit to the operating-system prompt.

Use the first method when you have only one command to enter; use the second method when you have several commands to enter.

Entering a single command from the debugger command line

If you need to enter only a single operating-system command, supply it as a parameter to the SYSTEM command. For example, in MS-DOS, if you want to copy a file from another directory into the current directory, you might enter:

system "copy a:\backup\sample.c sample.c" ⏎

If the operating-system command produces a display of some sort (such as a message), the debugger will blank the upper portion of the debugger display to show the information. In this situation, you can use the *flag* parameter to tell the debugger whether or not it should hesitate after displaying the results of the operating-system command. *Flag* may be a 0 or a 1:

0   The debugger immediately returns to the debugger environment after the last item of information is displayed.

1   The debugger does not return to the debugger environment until you press ⏎. (This is the default.)

In the example above, the debugger would open a system shell to display the following message:

```
         1 File(s) copied
Type Carriage Return To Return To Debugger
```

The message would be displayed until you pressed ⏎.

If you wanted the debugger to display the message and then return immediately to the debugger environment, you could enter the command in this way:

system ″copy a:\backup\sample.c sample.c″,0 ⏎

Entering several commands from a system shell

If you need to enter several commands, enter the SYSTEM command without parameters. The debugger will open a system shell and display the operating-system prompt. At this point, you can enter any operating-system command.

When you are finished entering commands and are ready to return to the debugger environment, enter the appropriate information:

    MS-DOS   *UNIX*

    exit ⏎   **exit** ⏎ **or** CONTROL D

Note:

On PC systems, available memory may limit the operating-system commands that you can enter from a system shell. For example, you would not be able to invoke another version of the debugger.

Additional system commands

The debugger also provides separate commands for changing directories and for listing the contents of a directory.

cd   Use the CHDIR (CD) command to change the current working directory. The format for this command is:

chdir   *directory name*

or   **cd**   *directory name*

This changes the current directory to the specified *directory name*. You can use relative pathnames as part of the directory name. Note that this command can affect any command whose parameter is a filename (such as the FILE, LOAD, and TAKE commands).

dir   Use the DIR command to list the contents of a directory. The format for this command is:

dir   [*directory name*]

This 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 this parameter, the debugger lists the contents of the current directory.

You can use wildcards as part of the *directory name*.

Defining a Memory Map

Before you begin a debugging session, you must supply the debugger with a memory map. The memory map tells the debugger which areas of memory it can and can't access. Note that the commands described in this chapter can also be entered using the Memory pulldown menu.

Synopsis Page		Topic
This chapter shows you how to set up a memory map for your system.	**8.1 The Memory Map: What It Is and Why You Should Define It**	ii
	8.2 Sample Memory Maps	iii
	8.3 Identifying Usable Memory Ranges	v
	8.4 Enabling Memory Mapping	vi
	8.5 Checking the Memory Map	vi
	8.6 Modifying the Memory Map During a Debugging Session	vii
	Returning to the original memory map	viii
	8.7 Using Multiple Memory Maps for Multiple Systems	ix

8.1 The Memory Map: What It Is and Why You Should Define It

A *memory map* tells the debugger which areas of memory it can and can't access. Memory maps vary, depending on the application; in general, the memory map should identify '340 local memory space plus additional memory space on your board. For the emulator, the memory map may correspond to the memory configuration of your target system. Typically, the map matches the MEMORY definition in your linker command file.

When memory mapping is enabled, the debugger checks each of its memory accesses against the provided memory map. If you attempt to access an undefined or protected area, the debugger displays an error message.

Note:

When the debugger compares memory accesses against the memory map, it performs this checking in software, not hardware. The debugger cannot prevent your program from attempting to access nonexistent memory.

The debugger provides a complete set of memory-mapping commands. You can define the memory map interactively by entering these commands while you're using the debugger. This can be inconvenient because, in most cases, you'll set up one memory map before you begin debugging and will use this map for all of your debugging sessions. The easiest method for doing this is to put the memory-mapping commands in a batch file.

Whenever you invoke the debugger, it looks for a special initialization batch file. The batch file name differs for the two versions of the debugger. For the emulator, this file is named *emuinit.cmd.* For the development boards, this file is named *dbinit.cmd.*

If the debugger finds the file, the debugger automatically reads and executes the commands in the file. If you plan to use the same memory map many times, it may be convenient for you to define your memory map in this batch file. However, you aren't required to use the default initialization batch file. You can use the –t debugger option to identify your own batch file.

The batch file shipped with the debugger disables memory mapping. This means that the debugger doesn't check to see if it is accessing valid memory; in effect, the entire memory range is treated as valid. Before beginning the debugging process in earnest, you will probably want to supply the debugger with a memory map and enable mapping.

If memory mapping is enabled but you have not defined a memory map, the debugger is initially unable to access memory. Invalid memory addresses and their contents are highlighted in the data-display windows. (On color monitors, invalid memory locations, by default, are displayed in red.)

8.2 Sample Memory Maps

This section contains sample memory maps for development boards and for the emulator. You can use the sample maps if you wish, but that is not necessary; the main purpose of showing them is to provide you with examples to use as a starting point. These samples correspond to two files that are shipped with the debugger, tdbmap.cmd and sdbmap.cmd:

❏ Figure 8–1 *(a)* shows the contents of tdbmap.cmd; Figure 8–1 *(a)* shows the memory map defined by the commands in tdbmap.cmd. This memory map can be used with a '34010 development board such as the '34010 TIGA development board.

❏ Figure 8–2 *(a)* shows the contents of sdbmap.cmd; Figure 8–2 *(a)* shows the memory map defined by the commands in sdbmap.cmd. This memory map can be used with the '34020 emulator or with a '34020 development board such as the '34020 software development board.

If you are using a third-party development board, you should refer to the board documentation for additional memory map information. If you don't find any information to help you set up a memory map, then you may want to leave memory mapping disabled.

Figure 8–1. Sample Memory Map for Use With a '34010 Development Board

(a) Memory map commands (tdbmap.cmd)   *(a) Memory map for '34010 local memory*

```
map on
ma 0x00002000,0x30,RW
ma 0x00003000,0x40,WRITEONLY
ma 0x10000000,0x00800000,RAM
ma 0xc0000000,0x200,RAM
ma 0xff800000,0x00800000,RAM
```

Address range	Memory type
0x0000 0000 to 0x0000 1FFF	reserved
0x0000 2000 to 0x0000 202F	control registers
0x0000 2030 to 0x0000 2FFF	reserved
0x0000 3000 to 0x0000 303F	palette registers
0x0000 3040 to 0x0FFF FFFF	reserved
0x1000 0000 to 0x107F FFFF	display memory
0x1080 0000 to 0xBFFF FFFF	reserved
0xC000 0000 to 0xC000 01FF	I/O registers
0xC000 0200 to 0xFF7F FFFF	reserved
0xFF80 0000 to 0xFFFF FFFF	program/data memory

Figure 8–2. *Sample Memory Map for Use With the '34020 Emulator
or a '34020 Development Board*

(a) Memory map commands (sdbmap.cmd)

```
map on
ma 0x00000000,0x00800000,RAM
ma 0x00800000,0x00800000,RAM
ma 0x01000000,0xbf000000,RAM
ma 0xc0000000,0x10000000,RAM
ma 0xd0000000,0x20,RAM
ma 0xd0000020,0x20,RAM
ma 0xd0000040,0x20,RAM
ma 0xd0000060,0x20,WRITEONLY
ma 0xd0000080,0x20,READONLY
ma 0xd00000a0,0x20,RAM
ma 0xd00000c0,0x20,PROTECT
ma 0xd00000e0,0x20,READONLY
ma 0xd0000100,0x0fffff00,RAM
ma 0xe00000000,0x20,RAM
ma 0xe0000020,0x0fffffe0,RAM
ma 0xf0000000,0x10000000,RAM
```

(a) Memory map for '34020 local memory

Address range	Description
0x0000 0000 to 0x007F FFFF	VRAM space
0x0080 0000 to 0x00FF FFFF	DRAM space
0x0100 0000 to 0xBFFF FFFF	aliased VRAM/DRAM addresses
0xC000 0000 to 0xCFFF FFFF	I/O registers
0xD000 0000 to 0xD000 001F	palette address register (RAM write mode)
0xD000 0020 to 0xD000 003F	color palette RAM
0xD000 0040 to 0xD000 005F	pixel read mask register
0xD000 0060 to 0xD000 007F	palette address (write mode)
0xD000 0080 to 0xD000 000F	palette address (read only)
0xD000 00A0 to 0xD000 00BF	overlay registers
0xD000 00C0 to 0xD000 00DF	reserved
0xD000 00E0 to 0xD000 00FF	palette address (register overlay read)
0xD000 0100 to 0xDFFF FFFF	aliased palette register addresses
0xE000 0000 to 0xEFFF FFFF	hardware space register
0xE000 0020 to 0xEFFF FFFF	aliased hardware space register addresses
0xF000 0000 to 0xFFFF FFFF	interrupt and TRAP vectors

8.3   **Identifying Usable Memory Ranges**

ma   The debugger's MA (memory add) command identifies valid ranges of target memory. The syntax for this command is:

ma   *address, length, type*

❑ The *address* parameter defines the starting address of a memory range. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.

❑ The *length* parameter defines the length, **in bits**, of the range. This parameter can be any C expression.

❑ The *type* parameter identifies the read/write characteristics of the memory range. The *type* must be one of these keywords:

To identify this kind of memory	Use this keyword as a *type* **parameter**
read-only memory	**R**, **ROM**, or **READONLY**
write-only memory	**W**, **WOM**, or **WRITEONLY**
read/write memory	**RW** or **RAM**
no-access memory	**PROTECT**

A new memory map must not overlap an existing entry. If you define a range that overlaps an existing range, the debugger ignores the new range and displays this error message in the COMMAND window display area:

```
Conflicting map range
```

Note:

When memory mapping is enabled, you cannot:

❑ Access memory locations that are not defined by an MA command
❑ Modify memory areas that are defined as read only or protected

If you attempt to access memory in these situations, the debugger displays this message in the COMMAND window display area:

```
Error in expression
```

8.4   Enabling Memory Mapping

map   By default, mapping is not enabled when you invoke the debugger. In order to use memory mapping, you must explicitly enable the memory mapping capability. Use the MAP command to do this; the syntax is:

map on
or   **map off**

Note that disabling memory mapping can cause bus fault problems in the target because the debugger may attempt to access nonexistent memory.

8.5   Checking the Memory Map

ml   If you want to see which memory ranges are defined, use the ML command. The syntax for this command is:

ml

The ML command lists the starting address, ending address, and read/write characteristics of each defined memory range. For example, if you're using the sample TDB memory map and you enter the ML command, the debugger displays this:

```
        Memory range            Attributes
        00002000 – 0000202f     READ WRITE
        00003000 – 0000303f     WRITE
        10000000 – 107fffff     READ WRITE
        ff800000 – ffffffff     READ WRITE

           │               │
      starting address   ending address
```

8.6   Modifying the Memory Map During a Debugging Session

If you need to modify the memory map during a debugging session, use these commands.

md   To delete a range of memory from the memory map, use the MD (memory delete) command. The syntax for this command is:

md   *address*

The *address* parameter identifies the starting address of the range of memory. If you supply an *address* that is not the starting address of a range, the debugger displays this error message in the COMMAND window display area:

```
Specified map not found
```

mr   If you want to delete all defined memory ranges from the memory map, use the MR (memory reset) command. The syntax for this command is:

mr

This resets the debugger memory map.

ma   If you want to add a memory range to the memory map, use the MA (memory add) command. The syntax for this command is:

ma   *address, length, type*

The MA command is described in detail on page v.

Returning to the original memory map

If you modify the memory map, you may want to go back to the original memory map without quitting and reinvoking the debugger. You can do this by resetting the memory map and then using the TAKE command to read in your original memory map from a batch file.

Suppose, for example, that you had set up your memory map in a batch file named *mem.map*. You could enter these commands to go back to this map:

mr ⏎   *Reset the memory map*
take mem.map ⏎   *Reread the default memory map*

The MR command resets the memory map. (Note that you could put the MR command in the batch file, preceding the commands that define the memory map.) The TAKE command tells the debugger to execute commands from the specified batch file.

8.7 Using Multiple Memory Maps for Multiple Systems

If you're debugging multiple applications, you may need a memory map for each target system. Here's the simplest method for handling this situation.

Step 1: Let the dbinit.cmd or emuinit.cmd file define the memory map for one of your applications.

Step 2: Create a separate batch file that defines the memory map for the additional system. The filename is unimportant, but for the purposes of this example, assume that the file is named *filename.x*. The general format of this file's contents should be

```
mr                                      Reset the memory map
MA commands                             Define the new memory map
map on                                  Enable mapping
```

(Of course, you can include any other appropriate commands in this batch file.)

Step 3: Invoke the debugger as usual.

Step 4: The debugger reads init.cmd during invocation. Before you begin debugging, read in the commands from the new batch file:

```
take filename.x ⏎
```

This redefines the memory map for the current debugging session.

Loading, Displaying, and Running Code

The main purpose of a debugging system is to allow you to load and run your programs in a test environment. This chapter tells you how to load your programs into the debugging environment, run them on the target system, and view the associated source code. Note that many of the commands described in this chapter can also be executed from the Load pulldown menu.

9.1 Code-Display Windows: Viewing Assembly Language Code, C Code, or Both

The debugger has three code-display windows:

❏ The DISASSEMBLY window displays the reverse assembly of program memory contents.

❏ The FILE window displays any text file; its main purpose is to display C source files.

❏ The CALLS window identifies the current function (when C code is running).

You can view code in several different ways. The debugger has three different code displays that are associated with the three debugging modes. The debugger selects the appropriate display based on two factors:

❏ The mode you select and
❏ Whether your program is currently executing assembly language code or C code.

Here's a summary of the modes and displays; for a complete description of the three debugging modes, refer to Section 6.1, *Debugging Modes and Default Displays* (page iii).

Use this mode	To view	The debugger uses these code-display windows
assembly mode	*assembly language code only* (even if your program is executing C code)	❏ DISASSEMBLY
auto mode	*assembly language code* (when that's what your program is running)	❏ DISASSEMBLY
auto mode	*C code only* (when that's what your program is running)	❏ FILE ❏ CALLS
mixed mode	*both assembly language and C code*	❏ DISASSEMBLY ❏ FILE ❏ CALLS

You can switch freely between the modes. If you choose auto mode, then the debugger displays C code *or* assembly language code, depending on the type of code that is currently executing.
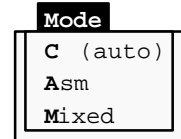
Selecting a debugging mode

When you first invoke the debugger, it automatically comes up in auto mode. You can then choose assembly or mixed mode. There are several ways to do this.

The Mode pulldown menu provides an easy method for switching modes. There are several ways to use the pulldown menus; here's one method:

Mode
C (auto)
Asm
Mixed

1) Point to the menu name.

2) Press the left mouse button; do not release the button. Move the mouse down the menu until your choice is highlighted.

3) Release the mouse button.

For more information about the pulldown menus, refer to Section 7.2, *Using the Pulldown Menus*, on page vii.

(F3)   Pressing this key causes the debugger to switch modes in this order:

auto ——————→ assembly ——————→ mixed
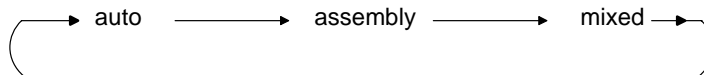
Enter any of these commands to switch to the desired debugging mode:

c   Changes from the current mode to auto mode.

asm   Changes from the current mode to assembly mode.

mix   Changes from the current mode to mixed mode.

If you are already in the desired mode when you enter a mode command, then the command has no effect.

9.2 Displaying Your Source Programs (or Other Text Files)

The debugger displays two types of code:

❑ It displays **assembly language code** in the DISASSEMBLY window in auto, assembly, or mixed mode.

❑ It displays **C code** in the FILE window in auto and mixed modes.

The DISASSEMBLY and FILE windows are primarily intended for displaying code that the PC points to. By default, the FILE window displays the C source for the current function (if any), and the DISASSEMBLY window shows the current disassembly.

Sometimes it's useful to display other files or different parts of the same file; for example, you may want to set a breakpoint at an undisplayed line. The DISASSEMBLY and FILE windows are not large enough to show the entire contents of most assembly language and C files. You can scroll through the windows. You can also tell the debugger to display specific portions of the disassembly or C source.

Displaying assembly language code

The assembly language code in the DISASSEMBLY window is the reverse assembly of program-memory contents. (This code doesn't come from any of your text files or from the intermediate assembly files produced by the compiler.)
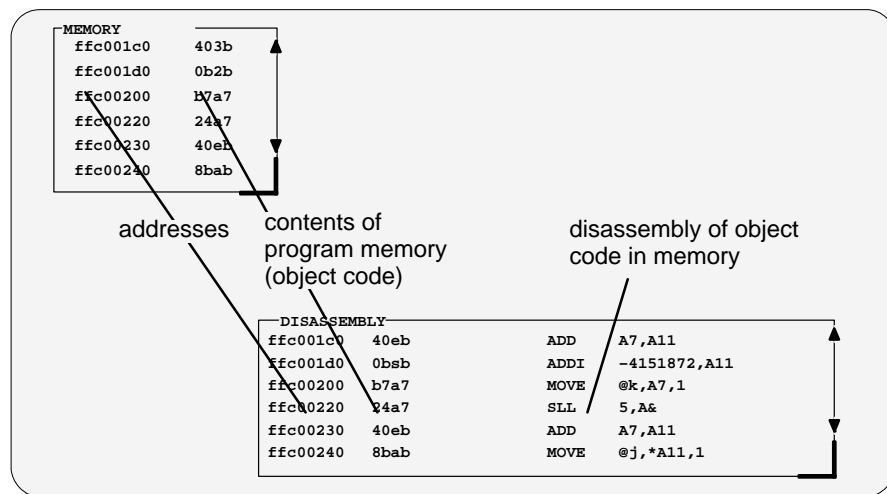
When you invoke the debugger, it comes up in auto mode. If you load an object file when you invoke the debugger, then the DISASSEMBLY window displays the reverse assembly of the object file that's loaded into memory. If you don't load an object file, the DISASSEMBLY window shows the reverse assembly of whatever happens to be in memory.

In assembly and mixed modes, you can use these commands to display a different portion of code in the DISASSEMBLY window.

dasm Use the DASM command to display code beginning at a specific point. The syntax for this command is:

dasm *address*
or **dasm** *function name*

This command modifies the display so that *address* or *function name* is displayed within the DISASSEMBLY window. The debugger continues to display this portion of the code until you run a program and halt it.

addr Use the ADDR command to display assembly language code beginning at a specific point. The syntax for this command is:

addr *address*
or **addr** *function name*

In assembly mode, ADDR works like the DASM command, positioning the code starting at *address* or at *function name* as the first line of code in the DISASSEMBLY window. In mixed mode, ADDR affects both the DISASSEMBLY and FILE windows.

Displaying C code

Unlike assembly language code, C code isn't reconstructed from memory contents—the C code that you view is your original C source. You can display C code explicitly or implicitly:

❑ You can force the debugger to show C source by entering a FILE, FUNC, or ADDR command.

❑ In auto and mixed modes, the debugger automatically opens a FILE window if you're currently running C code.

These commands are valid in C and mixed modes:

file   Use the FILE command to display the contents of any text file. The syntax for this command is:

    file   *filename*

This command uses the FILE window to display the contents of *filename*. The debugger continues to display this file until you run a program and halt in a C function. Although this command is most useful for viewing C code, you can use the FILE command for displaying any text file. You can view only one text file at a time. You can also access this command from the Load pulldown menu.

(Displaying a file *doesn't* load that file's object code. If you want to be able to run the program, you must load the file's associated object code as described in Section 9.3 on page viii.)

func   Use the FUNC command to display a specific C function. The syntax for this command is:

    func   *function name*
or   **func**   *address*

FUNC modifies the display so that *function name* or *address* is displayed within the window. If you supply an *address* instead of a *function name*, the FILE window displays the function containing *address* and places the cursor at that line.

Note that FUNC works similarly to FILE, but you don't need to identify the name of the file that contains the function.

addr Use the ADDR command to display C code beginning at a specific point. The syntax for this command is:

 addr *address*
or **addr** *function name*

In a C display, ADDR works like the FUNC command, positioning the code starting at *address* or at *function name* as the first line of code in the FILE window. In mixed mode, ADDR affects both the FILE and DISASSEMBLY windows.

Whenever the CALLS window is open, you can use the mouse or function keys to display a specific C function. This is similar to the FUNC or ADDR command but applies only to the functions listed in the CALLS window.

🖰 1) In the CALLS window, point to the name of C function.

🖱 2) Click the left mouse button.

(If the CALLS window is active, you can also use the arrow keys and F9 to display the function; see the *CALLS window* discussion on page xi for details.)

Displaying other text files

The DISASSEMBLY window always displays the reverse assembly of memory contents, no matter what is in memory.

The FILE window is primarily for displaying C code, but you can use the FILE command to display any text file within the FILE window. You may, for example, wish to examine system files such as autoexec.bat or init.cmd. You can also view your original assembly language source files in the FILE window.

You are restricted to displaying files that are 65,518 bytes long or less.

9.3 Loading Object Code

In order to debug a program, you must load the program's object code into memory. You can do this as you're invoking the debugger, or you can do it after you've invoked the debugger. (Note that you create an object file by compiling, assembling, and linking your source files; see Section 5.3, *Preparing Your Program for Debugging*, on page viii.)

Loading code while invoking the debugger

You can load an object file when you invoke the debugger (this has the same effect as using the debugger's LOAD command). To do this, enter:

development board: **emulator:**
db340   *object filename*   **db340emu**   *object filename*

If you want to load a file's symbol table only, use the –s option (this has the same effect as using the debugger's SLOAD command). To do this, enter:

development board: **emulator:**
db340 **–s** *object filename*   **db340emu** **–s** *object filename*

Loading code after invoking the debugger

After you invoke the debugger, you can use one of three commands to load object code and/or the symbol table associated with an object file. Use these commands as described below, or use them from the Load pulldown menu.

load   Use the LOAD command to load both an object file and its associated symbol table. In effect, the LOAD command performs both a RELOAD and an SLOAD. The format for this command is:

load   *object filename*

reload   Use the RELOAD command to load only an object file *without* loading its associated symbol table. This is useful for reloading a program when memory has been corrupted. The format for this command is:

reload   *object filename*

sload   Use the SLOAD command to load only a symbol table. The format for this command is:

sload   *object filename*

SLOAD is useful in a debugging environment in which the debugger cannot, or need not, load the object code (for example, if the code is in ROM). SLOAD clears the existing symbol table before loading the new one but does not modify memory or set the program entry point.

9.4 Where the Debugger Looks for Source Files

Some commands (FILE, LOAD, RELOAD, and SLOAD) expect a filename as a parameter. If the filename includes path information, the debugger uses the file from the specified directory and does not search for the file in any other directory. If you don't supply path information, though, the debugger must search for the file. The debugger first looks for these files in the current directory. You may, however, have your files in several different directories.

❑ If you're using LOAD, RELOAD, or SLOAD, you have only two choices for supplying the path information:

■ Specify the path as part of the filename.

cd   Alternatively, you can use the CD command to change the current directory from within the debugger. The format for this command is:

cd   *directory name*

❑ If you're using the FILE command, you have several options:

■ Within the DOS environment, you can name additional directories with the D_SRC environment variable. The format for doing this is:

SET D_SRC=*C:\pathname;C:\pathname*

This allows you to name several directories that the debugger can search. If you use the same directories often, it may be convenient to set the D_SRC environment variable in your autoexec.bat file. If you do this, then the list of directories is always available when you're using the debugger.

■ When you invoke the debugger, you can use the – i option to name additional source directories for the debugger to search. The format for this is:

development board:	**emulator:**
db340 –i *pathname*	**db340emu –i** *pathname*

You can specify multiple pathnames by using several –i options (one pathname per option). The list of source directories that you create with –i options is valid until you quit the debugger.

use   Within the debugger environment, you can use the USE command to name additional source directories. The format for this command is:

use   *directory name*

You can specify only one directory at a time.

In all cases, you can use relative pathnames such as ..\csource or ..\..\code. The debugger can recognize a cumulative total of 20 paths specified with D_SRC, –i, and USE.

9.5 Running Your Programs

To debug your programs, you must execute them on a '340 device. The debugger provides two basic types of commands to help you run your code:

❏ Basic **run commands** run your code without updating the display until you explicitly halt execution. There are several ways to halt execution:

  ■ Before you issue a run command, define a specific stopping point (for example, set a breakpoint).
  ■ Press ⌜ESC⌝.
  ■ Click the left mouse button.

❏ **Single-step** commands execute assembly language or C code, one statement at time, and update the display after each execution.

Defining the starting point for program execution

All run and single-step commands begin executing from the current PC (program counter). When you load an object file, the PC is automatically set to the starting point for program execution. You can easily identify the current PC by:

❏ Finding its entry in the CPU window

  or

❏ Finding the appropriately highlighted line in the FILE or DISASSEMBLY window. You can do this by executing one of these commands:

```
   dasm  PC
or addr  PC
```

Sometimes you may want to modify the PC to point to a different position in your program. There are two ways to do this:

rest   If you executed some code and would like to rerun the program from the original program entry point, use the RESTART (REST) command. The format for this command is:

  restart
  or **rest**

  Note that you can also access this command from the Load pulldown menu.

?/eval   You can directly modify the PC's contents with one of these commands:

?PC=_new value_

or   **eval pc =** _new value_

After halting execution, you can continue from the current PC by reissuing any of the run or single-step commands.

Running code

The debugger supports several run commands.

run   The RUN command is the basic command for running an entire program. The format for this command is:

run   [_expression_]

The command's behavior depends on the type of parameter you supply:

❑ If you don't supply an _expression_, the program executes until it encounters a breakpoint or until you press ⌈ESC⌋ or the left mouse button.

❑ If you supply a logical or relational _expression_, this becomes a conditional run (see page xv).

❑ If you supply any other type of _expression_, the debugger treats the expression as a _count_ parameter. The debugger executes _count_ instructions, halts, then updates the display.

⌈F5⌋   Pressing this key runs code from the current PC. This is similar to entering a RUN command without an _expression_ parameter.

go   Use the GO command to execute code up to a specific point in your program. The format for this command is:

go   [_address_]

If you don't supply an _address_ parameter, then GO acts like a RUN command without an _expression_ parameter.

ret   The RETURN (RET) command executes the code in the current C function and halts when execution returns to its caller. The format for this command is:

return

or   **ret**

Breakpoints do not affect this command, but you can halt execution by pressing (ESC) or the left mouse button.
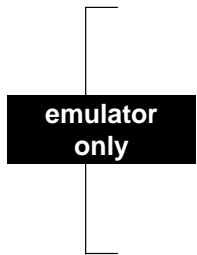
runb   Use the RUNB (run benchmark) command to execute a specific section of code and count the number of clock cycles consumed by the execution. The format for this command is:

emulator only

runb

Using the RUNB command to benchmark code is a multistep process, described later in this chapter (Section 9.7, *Benchmarking*, on page xvii).

Single-stepping through code

Single-step execution is similar to running a program that has a breakpoint set on each line. The debugger executes one statement, updates the display, and halts execution. (You can supply a parameter that tells the debugger to single-step more than one statement; the debugger updates the display after each statement.) You can single-step through assembly language code or C code.

The debugger supports several commands for single-stepping through a program. Command execution may vary, depending on whether you're single-stepping through C code or assembly language code.

Note that the debugger ignores interrupts when you use the STEP command to single-step through assembly language code.

Each of the single-step commands has an optional *expression* parameter that works like this:

❑ If you don't supply an *expression*, the program executes a single statement, then halts.

❑ If you supply a logical or relational *expression*, this becomes a conditional single-step execution (see page xv).

❑ If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger single-steps *count* C or assembly language statements (depending on the type of code you're in).

step   Use the STEP command to single-step through assembly language or C code. The format for this command is:

step   [*expression*]

If you're in C code, the debugger executes one C statement at a time. In assembly or mixed mode, the debugger executes one assembly language statement at a time.

If you're single-stepping through C code and encounter a function call, the STEP command shows you the single-step execution of the called function (assuming that the function was compiled with the compiler's –g debug option). When function execution completes, single-step execution returns to the caller. If the function wasn't compiled with the debug option, the debugger executes the function but doesn't show single-step execution of the function.

cstep The CSTEP command is similar to STEP, but CSTEP always single-steps in terms of a C statement. If you're in C code, STEP and CSTEP behave identically. In assembly language code, however, CSTEP executes all assembly language statements associated with one C statement before updating the display. The format for this command is:

> **cstep** [*expression*]

next The NEXT and CNEXT commands are similar to the STEP and CSTEP
cnext commands. The only difference is that NEXT/CNEXT never show single-step execution of called functions—they always step to the next consecutive statement. The formats for these commands are:

> **next** [*expression*]
> **cnext** [*expression*]

You can also single-step through programs by using function keys:

⌨ F8 Acts as a STEP command.

⌨ F10 Acts as a NEXT command.

The debugger allows you to execute several single-step commands from the selections on the menu bar.

To execute a STEP:

1) Point to `Step=F8` in the menu bar.

2) Click the left mouse button.

To execute a NEXT:

1) Point to `Next=F10` in the menu bar.

2) Click the left mouse button.

Running code while disconnected from the target

runf Use the RUNF command to disconnect the emulator from the target system while code is executing. The format for this command is:

runf

When you enter RUNF, the debugger clears all breakpoints, disconnects the emulator from the target system, and causes the processor to begin execution at the current PC. You can quit the debugger, or you can continue to enter commands. However, any command that causes the debugger to access the target at this time will produce an error.

RUNF is useful in a multiprocessor system. It's also useful in a system in which several target systems share an emulator; RUNF enables you to disconnect the emulator from one system and connect it to another.

emulator only

halt Use the HALT command to halt the target system after you've entered a RUNF command. The format for this command is:

halt

When you invoke the debugger, it automatically executes a HALT command. Thus, if you enter a RUNF, quit the debugger, and later reinvoke the debugger, you will effectively reconnect the emulator to the target system and run the debugger in its normal mode of operation. When you invoke the debugger, use the –s option to preserve the current PC and memory contents.

reset The RESET command behaves differently for the two versions of the debugger:

❑ For the development board version of the debugger, RESET reloads the monitor (gspmon) but does not reset the '340 processor.

❑ For the emulator version of the debugger, RESET resets the target system. This is a *software* reset.

The format for this command is:

reset

Running code conditionally

The RUN, GO, and single-step commands have an optional *expression* parameter that can be a relational or logical expression. This type of expression has one of the following operators as the highest precedence operator in the expression:

>	> =	<
< =	= =	! =
&&	| |	!

When you use this type of expression with these commands, the command becomes a conditional run. The debugger executes the command repeatedly for as long as the expression evaluates to true.

You must use breakpoints with conditional runs; the expression is evaluated each time the debugger encounters a breakpoint. Each time the debugger evaluates the conditional expression, it updates the screen. The debugger applies this algorithm:

top:
   if (*expression* = = 0) go to end;
   run or single-step (until breakpoint, ᴱˢᶜ , or mouse button halts execution)
   if (halted by breakpoint, *not* by ᴱˢᶜ or mouse button) go to top
end:

Generally, you should set the breakpoints on statements that are related in some way to the expression. For example, if you're watching a particular variable in a WATCH window, you may want to set breakpoints on statements that affect that variable and use that variable in the expression.

9.6 Halting Program Execution

Whenever you're running or single-stepping code, program execution halts automatically if the debugger encounters a breakpoint or if it reaches a particular point where you told it to stop (by supplying a *count* or an *address*). If you'd like to explicitly halt program execution, there are two ways to accomplish this:

〕 Click the left mouse button.

ⓔⓢⓒ Press the escape key.

After halting execution, you can continue program execution from the current PC by reissuing any of the run or single-step commands.

9.7 **Benchmarking**

The debugger allows you to keep track of the number of CPU clock cycles consumed by a particular section of code. The debugger maintains the count in a pseudoregister named *CLK*.

Note:

The value in CLK is valid only after using a RUNB command that is terminated by a breakpoint.

Benchmarking code is a multiple-step process:

emulator only

Step 1: Set a breakpoint at the statement that marks the beginning of the section of code you'd like to benchmark.

Step 2: Set a breakpoint at the statement that marks the end of the section of code you'd like to benchmark.

Step 3: Enter any RUN command to execute code up to the first breakpoint.

Step 4: Now enter the RUNB command:

runb ⏎

When the processor halts at the second breakpoint, the value of CLK is valid. To display it, use the ? command or enter it into the WATCH window with the WA command. This value is valid until you enter another RUN command.

Loading TIGA Applications

If you are working with a TIGA application containing modules you have written yourself, you may want to use the debugger to refine these modules. Conventional debugging strategies are not adequate for this situation.

Since TIGA downloads and locates modules at runtime, under usual circumstances, the modules would not be available to the debugger. However, TIGA and the debugger support a special feature which informs the debugger when a module has been downloaded. Thus, allowing you to set a breakpoint at the entry of your module. Therefore, when TIGA downloads your module, the TIGA application halts and you have control of the module for debugging purposes. This feature is called *dynamic loading*. Once a module has been dynamically loaded, you can use all of the basic debugger features to debug your module.

Topic	Page

This chapter discusses the process of setting and clearing tentative breakpoints in TIGA applications.

10.1 Overview of the Dynamic-Load Process

Below are the procedures you should follow to use dynamic loading:

Step 1: Be sure you have followed the installation procedures in the *Setting Up for TIGA Applications* section in the appropriate installation chapter:

❏ Be sure that you have installed the TIGA communications driver (tigacd) on the target PC.

emulator: **tigacd /d1** ⏎

development boards: **tigacd /d2** ⏎

Remember, you must install tigacd whenever you power up or reboot the target PC.

❏ Be sure you have copied the RLMs to the host PC (so that they're on both PCs). In addition, you may want to copy the C source files to the host PC so that you'll be able to view them during debugging.

❏ Be sure you have copied tigagm.out and, if you are using TIGA extended primitives, copy extprims.rlm to the directory on the target PC containing tigacd. Also copy these files to the directory on the host PC containing the debugger executable file.

❏ *Development boards only:* Be sure you have installed the serial communication drivers—debugcom on the host PC and tigacom on the target PC:

debugcom: **debugcom** [**C***communication port*] [**B***baud rate]*

tigacom: **tigacom** [**C***communication port*] [**B***baud rate]*

Remember, you must install these drivers whenever you power up or reboot the host or target PC.

Be sure that the serial cable pinouts are correct.

Step 2: Start the debugger on the host system.

Step 3: Set *tentative breakpoints* on the TIGA module functions you wish to debug. (Tentative breakpoints are a special type of breakpoint used only for identifying TIGA module functions that will be dynamically loaded. For more information, see page iv.)

Step 4: Start the TIGA application on the target system.

Step 5: As the TIGA application executes, TIGA will load code into the '340 board. When a module is loaded into '340 memory, the module will automatically be loaded into the debugger ("dynamically loaded"). When the breakpoint is reached, the TIGA application will halt.

Step 6: Now you can debug the module just as if it were any other type of code.

Step 7: When you have finished done debugging the module, run past the end of the module; this will restart the TIGA application.

Debugging with Microsoft Windows (version 3.0)

The dynamic-load capability is also useful for debugging the TIGA graphics manager (tigagm.out) and the Windows driver engine (win30.rlm or tigawin.rlm) while running Microsoft Windows. You can also debug TIGA extensions that are downloaded by a Windows application. However, there are a few restrictions and potential problems:

❏ The debugger and serial link driver will work *only* when Windows is run in real (win /r) or standard (win /s) modes.

❏ Be sure the Windows driver engine (win30.rlm or tigawin.rlm) is copied into the directory where the debugger executable file is located (on the host PC). Also, if you plan to debug downloaded extensions, the corresponding .rlm files must reside in the same directory as the debugger executable file.

❏ You may need to set Windows to use the keyboard rather than a mouse. If windows is set up for use with a mouse, it will poll the available serial ports, looking for a mouse. This may corrupt the serial port settings used by DEBUGCOM.

10.2 Setting a Tentative Breakpoint

Tentative breakpoints are a special type of breakpoint used only for identifying TIGA modules that will be dynamically loaded. In this case, the debugger identifies the breakpoints as unresolved symbols. Unresolved symbols are treated as empty variables within the current symbol table.

tba   The TBA command tells the debugger to look for your custom functions and set tentative breakpoints at their entry points. The basic syntax for this command is:

tba   *function name*

The tentative breakpoint command adds the function name to a list of tentative breakpoints and then evaluates the address. If a valid address is found, a breakpoint is set immediately. Otherwise, when a TIGA module is loaded into the debugger, the debugger processes the tentative breakpoint list by attempting to find the function name in the new symbol table. Breakpoints will be set at the addresses of the new modules.

Clearing a tentative breakpoint

tbd   Once set, you may delete tentative breakpoints from functions that have not yet been loaded by using the TBD command. The basic syntax for this command is:

tbd   *breakpoint index*

The *breakpoint index* parameter corresponds to index numbers displayed next to the tentative breakpoints listed with the BL debugger command (refer to page v for more details).

Note:

If a tentative breakpoint is resolved and active, it cannot be deleted with this command. You must use the BD or BR command as explained on page v.

Using regular breakpoint commands while debugging TIGA modules

The debugger treats the dynamic-load feature and its tentative breakpoints separately from other debugger features. However, you can use regular breakpoint commands while debugging a dynamically loaded TIGA module. Effects of using regular debugger commands with the TBA command are listed below:

❏ The **BL** command lists both regular breakpoints and tentative breakpoints.The tentative breakpoint list consists of any unresolved symbols entered with the TBA command. Breakpoints are identified by index numbers so that you can delete them with the TBD command if you want.

❏ The **BD** command deletes only *active* breakpoints. Therefore, any breakpoint that is unresolved at execution of the BD command and was originally set through the TBA command will be left on the tentative breakpoint list.

❏ The **BR** command deletes all regular and tentative breakpoints (unresolved as well as resolved).

❏ If you are using **RUNB** (benchmarking) and you load a TIGA module, you will receive inaccurate benchmark results.

10.3 Reloading TIGA Modules

The debugger maintains a list of the TIGA modules that have been dynamically loaded. The last module loaded is the default module. Any of the modules in this list can be selected again.

mod   The MOD command allows you to select a TIGA module that has been dynamically loaded. The basic syntax for this command is:

mod   [*TIGA module name*]

The module you select becomes the default module. This insures that the default module is searched first and thereby helps prevent misinterpretation of identical variable names in multiple applications.

To obtain a list of all the modules that have been dynamically selected, enter the MOD command with no parameter. This will also tell you which module is the default module and identify any code that has been loaded with the LOAD command.

Using LOAD, RELOAD, SLOAD, and RESTART while debugging TIGA modules

Do not use the LOAD, RELOAD, and RESTART commands. Doing so will interfere with the TIGA application and you will have to restart your application.

You can, however, use the SLOAD command to load a module's symbol table.

10.4 Identifying Symbols Used in TIGA Modules

Multiple TIGA modules may use symbols with the same names. The debugger provides a naming system to help you identify these symbols. The basic syntax for identifying symbols is:

[**{**module name**}**] function[**.**variable]

For example, assume you have a TIGA module named *draw* containing a function *circle*. In circle, you have a variable named *radius* that you want to watch. To do so, enter:

wa {draw} circle.radius ⏎

If you choose not to specify a module name, all applications are checked, beginning with the default (current) module. The first function found with the specified name is used. Note that module names are case sensitive.

Note:

Variables contained in your TIGA application will replace *duplicate* variable names in your current symbol table.

Managing Data

The debugger allows you to examine and modify many different types of data related to the target system and to your program. You can display and modify the values of:

> Individual memory locations or a range of memory
> '340 registers
> Variables, including scalar types (ints, chars, etc.) and aggregate types (arrays, structures, etc.)

This chapter tells you how to display and change data.

| | **Synopsis** | **Topic** |
| | **Page** | |

11.1 Where Data Is Displayed

Four windows are dedicated to displaying the various types of data.

Type of data	Window name and purpose
memory locations	**MEMORY window** Displays the contents of a range of data memory or program memory
register values	**CPU window** Displays the contents of '340 registers
pointer data or selected variables of an aggregate type	**DISP windows** Display the contents of aggregate types and show the values of individual members
selected variables (scalar types or individual members of aggregate types) and specific memory locations or registers	**WATCH window** Displays selected data

This group of windows is referred to as **data-display windows**.

11.2 Basic Commands for Managing Data

The debugger provides special-purpose commands for displaying and modifying data in dedicated windows. The debugger also supports several general-purpose commands that you can use to display or modify any type of data.

whatis If you want to know the type of a variable, use the WHATIS command. The syntax for this command is:

whatis *symbol*

This lists s*ymbol'*s data type in the COMMAND window display area. The *symbol* can be any variable (local, global, or static), a function name, structure tag, typedef name, or enumeration constant.

Command	Result displayed in the COMMAND window
`whatis giant`	`struct zzz giant[100];`

`whatis xxx`	`struct xxx      {`
	`    int a;`
	`    int b;`
	`    int c;`
	`    int f1 : 2;`
	`    int f2 : 4;`
	`    struct xxx *f3;`
	`    int f4[10];`
	`}`

? The ? (evaluate expression) command evaluates an expression and shows the result in the COMMAND window display area. The basic syntax for this command is:

? *expression*

The *expression* can be any C expression, including an expression with side effects. However, you cannot use a string constant or function call in the *expression.*

If the result of *expression* is scalar, then the debugger displays the result as a decimal value in the COMMAND window. If *expression* is a structure or array, ? displays the entire contents of the structure or array; you can halt long listings by pressing (ESC).

Here are some examples that use the ? command.

Command	Result displayed in the COMMAND window

`? giant`	`giant[0].b1 436547877`
	`giant[0].b2 –791051538`
	`giant[0].b3 1952557575`
	`giant[0].b4 –1555212096`
	etc.

`? j`	`4194425`
`? j=0x5a`	`90`

Note that the DISP command (described in detail on page xvii) behaves like the ? command when its *expression* parameter does not identify an aggregate type.

eval   The EVAL (evaluate expression) command behaves like the ? command *but does not show the result* in the COMMAND window display area. The syntax for this command is:

    eval   *expression*

or   **e**   *expression*

EVAL is useful for assigning values to registers or memory locations in a batch file (where it's not necessary to display the result).

11.3 Basic Methods for Changing Data Values

The debugger provides you with a great deal of flexibility in modifying various types of data. You can use the debugger's overwrite editing capability, which allows you to change a value simply by typing over its displayed value. You can also use the data-management commands for more complex editing.

Editing data displayed in a window

Use overwrite editing to modify data in a data-display window; you can edit:

❑ Registers displayed in the CPU window
❑ Memory contents displayed in the MEMORY window
❑ Elements displayed in a DISP window
❑ Values displayed in the WATCH window

There are two similar methods for overwriting displayed data:

This method is sometimes referred to as the "click and type" method.

↖ 1) Point to the data item that you want to modify.

▌⫶ 2) Click the left button. The debugger highlights the selected field. (Note that the window containing this field becomes active when you press the mouse button.)

ESC 3) Type the new information. If you make a mistake or change your mind, press ESC or move the mouse outside the field and press/release the left button; this resets the field to its original value.

↵ 4) When you finish typing the new information, press ↵ or any arrow key. This replaces the original value with the new value.

1) Select the window that contains the field you'd like to modify; make this the active window. (Use the mouse, the WIN command, or F6. For more detail, see Section 6.4, *The Active Window*, on page xx.)

2) Use arrow keys to move the cursor to the field you'd like to edit.

⬆ Moves up 1 field at a time.
⬇ Moves down 1 field at a time.
⬅ Moves left 1 field at a time.
➡ Moves right 1 field at a time.

F9   3) When the field you'd like to edit is highlighted, press F9 . The debugger highlights the field that the cursor is pointing to.

ESC   4) Type the new information. If you make a mistake or change your mind, press ESC ; this resets the field to its original value.

⏎   5) When you finish typing the new information, press ⏎ or any arrow key. This replaces the original value with the new value.

Advanced "editing"—using expressions with side effects

Using the overwrite editing feature to modify data is straightforward. However, there are other methods that take advantage of the fact that most debugger commands accept C expressions as parameters, and that C expressions can have *side effects*. When an expression has a side effect, it means that the value of some variable in the expression changes as the result of evaluating the expression.

This means that you can coerce many commands into changing values for you. Specifically, it's most helpful to use ? and EVAL to change data as well as display it.

For example, if you want see what's in register A3, you can enter:

```
? A3
```

You can also use this type of command to modify A3's contents. Here are some examples of how you might do this:

```
? A3++                    Side effect: increments the contents of A3 by 1
eval --A3                 Side effect: decrements the contents of A3 by 1
? A3 = 8                            Side effect: sets A3 to 8
eval A3/=2                 Side effect: divides contents of A3 by 2
```

Note that not all expressions have side effects. For example, if you enter **? A3+4**, the debugger displays the result of adding 4 to the contents of A3 but does not modify A3's contents. Expressions that have side effects must contain an assignment operator or an operator that implies an assignment. Operators that can cause a side effect are:

=	+=	−=	*=	/=
%=	&=	^=	|=	<<=
	>>=	++	−−	

11.4 Managing Data in Memory

In mixed and assembly modes, the debugger maintains a MEMORY window that displays the contents of memory. For details concerning the MEMORY window, see the *MEMORY window* discussion (page xiii).

```
        ┌─MEMORY─
        ffc45a80  ffff  ffff  ffff  ffff  ffff  ffff
        ffc45ae0  ffff  ffff  b460  ffff  0010  c000
        ffc45b40  0000  0000  b460  ffff  0010  c000
addresses       ffc45ba0  0010  c000  0000  0000  0080  0000       data
        ffc45c00  0000  0000  0010  2000  b080  ffff
        ffc45cc0  0010  c000  0000  0000  0080  0000
```
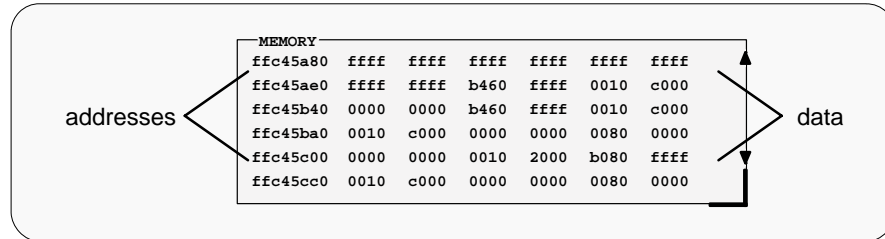
The debugger has commands that show the value at a specific location or display a different range. The debugger allows you to change the values at individual locations; refer to Section 11.3, *Basic Methods for Changing Data Values* (page vi), for more information.

Note that by default, the memory window shows 16-bit values beginning at even 16-bit boundaries. Because the '340 devices are bit addressable, you can display 16-bit values beginning at any bit boundary.

Displaying memory contents

The main way to observe memory contents is to view the display in the MEMORY window. The amount of memory that you can display is limited by the size of the MEMORY window (which is limited only by the screen size). During a debugging session, you may need to display different areas of memory within the window. The debugger provides two methods for doing this.

mem If you want to display a different memory range in the MEMORY window, use the MEM command. The basic syntax for this command is:

mem *expression*

This makes *expression* the first entry in the MEMORY window. The end of the range is defined by the size of the window: to show more memory locations, make the window larger (see *Resizing a window*, page xxiii, for more information).

The *expression* can be an absolute address, a symbolic address, or any C expression. Here are several examples:

❏ *Absolute address.* Suppose that you want to display memory beginning from the very first address. You might enter this command:

mem 0x00

Hint: MEMORY window addresses are shown in hexadecimal format. If you want to specify a hex address, be sure to prefix the address number with **0x**.

❏ *Symbolic address.* You can use any defined C symbol as an *expression* parameter. For example, if your program defined a symbol named *SYM*, you could enter this command:

mem &SYM

Hint: Prefix the symbol with the & operator to use the address of the symbol.

❏ *C expression.* If you use a C expression as a parameter, the debugger evaluates the expression and uses the result as a memory address:

mem SP – A0 + label

You can also change the display of any data-display window—including the MEMORY window—by scrolling through the window's contents. See the *Scrolling through a window's contents* discussion (page xxix) for more details.

Displaying memory contents while you're debugging C

If you're debugging C code in auto mode, you won't see a MEMORY window—the debugger doesn't show the MEMORY window in the C-only display. However, there are several ways to display memory in this situation.

Hint: If you want to use the *contents* of an address as a parameter, be sure to prefix the address with the C indirection operator (*).

❏ If you have only a temporary interest in the contents of a specific memory location, you can use the ? command to display the value at this address. For example, if you want to know the contents of memory location 20 (hex), you could enter:

```
? *0x20
```

The debugger displays the memory value in the COMMAND window display area.

❏ If you want the opportunity to observe a specific memory location over a longer period of time, you can display it in a WATCH window. Use the WA command to do this:

```
wa *0x20
```

❏ You can also use the DISP command to display memory contents. The DISP window shows memory in an array format with the specified address as "member" [0]. In this situation, you can also use casting to display memory contents in a different numeric format:

```
disp *(float *)0x20
```

Saving memory values to a file

ms   Sometimes it's useful to save a block of memory values to a file. You can use the MS (memory save) command to do this; the files are saved in COFF format. (For more information about COFF, refer to the *TMS340 Family Code-Generation Tools User's Guide.*) The syntax for the MS command is:

ms   *address, length, filename*

❑ The *address* parameter identifies the first address in the block.

❑ The *length* parameter defines the length, in words, of the block. This parameter can be any C expression.

❑ The *filename* is a system file.

For example, to save the values in data memory locations 0x0–0x10 to a file named memsave, you could enter:

ms 0x0,0x10,memsave ⏎

To reload memory values that were saved in a file, use the LOAD command. For example, to reload the values that were stored in memsave, enter:

load memsave ⏎

Filling a block of memory

fill   Sometimes it's useful to be able to fill an entire block of memory at once. You can do this by using the FILL command. The syntax for this command is:

fill   *address, length, data*

❏ The *address* parameter identifies the first address in the block.
❏ The *length* parameter defines the number of 32-bit words to fill.
❏ The *data* parameter is the value that is placed in each word in the block.

For example, to fill locations 0xFF80 0000 to 0xFF80 0300 with the value 0x1234 ABCD, you would enter:

`fill 0xff800000,0x18,0x1234abcd` ⏎

If you want to check to see that memory has been filled as you have asked, you can enter:

`mem 0xff800000` ⏎

This changes the MEMORY window display to show the block of memory beginning at address 0xFF80 0000.

Note that the FILL command can also be executed from the Memory pulldown menu.

11.5 Managing Register Data

In mixed and assembly modes, the debugger maintains a CPU window that displays the contents of the PC, the ST, and the A and B registers. For details concerning the CPU window, see the *CPU window* discussion (page xiv).

```
        ┌CPU
        PC    ffc00000    ST    20000010    A0    00000000
register A1    00000080    A2    00000000    A3    c0000010
name    A4    00000000    A5    00000000    A6    00000000
        A7    00000040    A8    00000000    A9    ffc45da0
        A10   ffffffff    A11   00001795    A12   00000000
        A13   ffc363e0    A14   ffc363e0    A15   ffc45d80
register B0    ffffffff    B1    00000000    B2    00000000
contents B3   00000000    B4    00000000    B5    00000000
        B6    00000000    B7    00000000    B8    00000000
        B9    00000000    B10   00000000    B11   00000000
        B12   00000000    B13   00000000    B14   00000000
```

In addition to the registers listed in the CPU window, you can display and modify the contents of I/O registers, selected status bits. and '34082 registers. You can also access register A15/B15 as *SP.* Note that the B-file registers cannot be accessed by name unless your program defines them as symbols.

The debugger provides commands that allow you to display and modify the contents of registers. You can use the debugger's overwrite editing capability to modify the contents of any register or bit displayed in the CPU, I/O, FPU, or WATCH window. You can use the data-management commands to modify the CPU registers as well as registers that are not displayed in the CPU window. Refer to Section 11.3, *Basic Methods for Changing Data Values* (page vi), for more information.

Displaying register contents

The main way to observe register contents is to view the display in the CPU window. However, you may not be interested in all of the registers; if you're interested in only a few registers, you might want to make the CPU window small and use the extra screen space for the DISASSEMBLY or FILE display. In this type of situation, there are several ways to observe the contents of the selected registers.

❑ If you have only a temporary interest in the contents of a register, you can use the ? command to display the register's contents. For example, if you want to know the contents of the SP, you could enter:

? SP

The debugger displays the SP's current contents in the COMMAND window display area.

❏ If you want to observe a register over a longer period of time, you can use the WA command to display the register in a WATCH window. For example, if you want to observe the status register, you could enter:

WA ST,Status Reg

This adds the ST to the WATCH window and labels it as *Status Reg*. The register's contents are continuously updated, just as if you were observing the register in the CPU window.

These methods are also useful when you're debugging C in auto mode, because the debugger doesn't show the CPU window in the C-only display.

Displaying and changing the contents of I/O registers

The simplest way to observe I/O register contents is to open the I/O window. To do this, use the IOREGS command:

ioregs ⏎

You can also access individual I/O registers by name (names are not case sensitive). All registers are 16 bits except BSFLTD, DPYNX, DPYST, HSTADR, and PMASK. (Note that PMASK is a 16-bit register for the '34010 and a 32-bit register for the '34020.) You can access the 16 LSBs of a 32-bit register by appending an L suffix to the register name; you can access the 16 MSBs by appending an H suffix to the register name.

Note:

Some of the registers displayed in the I/O window may be available only for the '34020. Additionally, for the '34010, you can access the REFCNT register as REFADR.

You can display or modify the contents of the I/O registers by using the ?, EVAL, and WA commands.

❏ The simplest way to display the contents of one of these registers is to use the ? command. For example, to display the contents of the PSIZE register, enter:

? PSIZE ⏎

❏ To change the contents of a register, use ? or EVAL. For example, if you want to set the pixel size to 4, enter:

? PSIZE = 4 ⏎

❏ If you want to observe a register value on a regular basis but don't want to open the I/O window, you can use the WA command to add the register value to the WATCH window. For example,

WA PSIZE,Pixel Size ⏎

Displaying and changing the contents of status bits

You can display or modify the following register bits:

Bit name	Access as	Bit name	Access as
CD (cache disable)	**CD**	CF (cache flush)	**CF**
HLT (halt)	**HLT**	IE (global interrupt enable)	**IE**
V (overflow)	**STV**	Z (zero)	**STZ**
C (carry)	**STC**	N (negative)	**STN**
FS0 (field size 0)	**FS0**	FE0 (field extension 0)	**FE0**
FS1 (field size 1)	**FS1**	FE1 (field extension 1)	**FE1**

Note:

❑ Bit names are not case sensitive.

❑ Some bits can be accessed by name; others must be prefixed with ST. Refer to the **Access as** column (above).

❑ For the '34020, you can also access the RST bit of the HSTCTLH register. (If you're using a development board, don't modify RST.)

When you are debugging assembly language routines, you may find it helpful to add the status bits to the WATCH window. This enables you to observe the processor state while running code.

Displaying and changing the contents of '34082 registers

If you invoked the debugger with the –mc option, you can access '34082 registers by opening the FPU window. To do this, use the FPUREGS command:

fpuregs ⏎

You can also access these registers by name (for example, RA0, RA1, etc.); they will be represented as double-precision scientific notation. You can display integer and float representations of these registers by replacing the *R* in the register name with *I* or *F*, respectively. For example:

❑ To display the integer value of RA0, enter:

? IA0 ⏎

❑ To watch the float version of RB3, enter:

wa FB3 ⏎

❑ You can also use the register values in expressions:

eval RA0 + FB5 ⏎

In this case, the addition is done in double precision; the debugger converts the float value in RB5 to a double.

11.6 Managing Data in a DISP (Display) Window

The main purpose of the DISP window is to display members of complex, aggregate data types such as arrays and structures. The debugger shows DISP windows *only when you specifically request to see DISP windows* with the DISP command (described below). Note that you can have up to 120 DISP windows open at once. For additional details about DISP windows, see the *DISP window* discussion (page xvii).
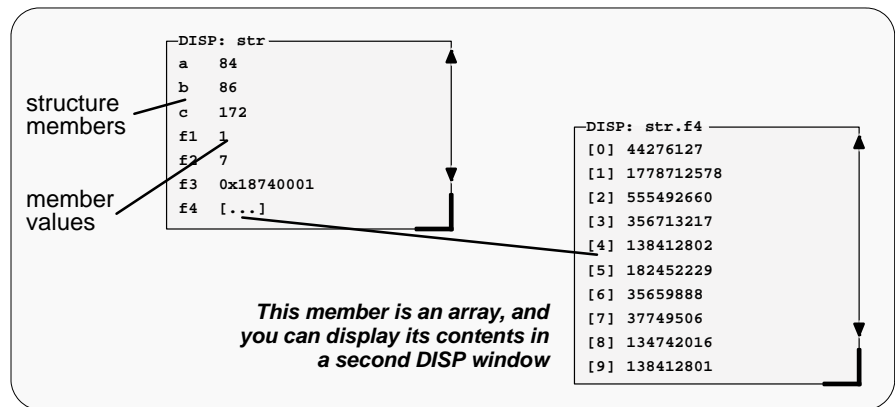
```
┌─DISP: str─────────
a    84
b    86
c    172
f1   1
f2   7
f3   0x18740001
f4   [...]
```

```
┌─DISP: str.f4──────
[0] 44276127
[1] 1778712578
[2] 555492660
[3] 356713217
[4] 138412802
[5] 182452229
[6] 35659888
[7] 37749506
[8] 134742016
[9] 138412801
```

structure members

member values

This member is an array, and you can display its contents in a second DISP window

Remember, you can use the data-management commands or the debugger's overwrite editing capability to modify the contents of any value displayed in a DISP window. Refer to Section 11.3, *Basic Methods for Changing Data Values* (page vi), for more information.

Displaying data in a DISP window

disp   To open a DISP window, use the DISP command. The basic syntax for this command is:

disp   *expression*

If the *expression* is not an array, structure, or pointer (of the form **pointer name*), the DISP command behaves like the ? command. However, if *expression* **is** one of these types, the debugger opens a DISP window to display the values of the members.

If a DISP window contains a long list of members, you can use PAGE DOWN, PAGE UP, or arrow keys to scroll through the window. If the window contains an array of structures, you can use CONTROL PAGE DOWN and CONTROL PAGE UP to scroll through the array.

Once you open a DISP window, you may find that a displayed member is another one of these types. This is how you identify the members that are arrays, structures, or pointers:

A member that is an array looks like this   [. . .]
A member that is a structure looks like this   {. . .}
A member that is a pointer looks like an address   0x00000000

You can display the additional data (the data pointed to or the members of the array or structure) in additional DISP windows (these are referred to as *children*). There are three ways to do this.

Use the DISP command again; this time, *expression* must identify the member that has additional data. For example, if the first expression identifies a structure named *str* and one of str's members is an array named *f4*, you can display the contents of the array by entering this command:

```
disp str.f4
```

This opens a new DISP window that shows the contents of the array. If str has a member named *f3* that is a pointer, you could enter:

```
disp *str.f3
```

This opens a window to display what str.f3 points to.

Here's another method of displaying the additional data:

1) Point to the member in the DISP window.

2) Now click the left button.

Here's the third method:

1) Use the arrow keys to move the cursor up and down in the list of members.

2) When the cursor is on the desired field, press F9 .

When the debugger opens a second DISP window, the new window may at first be displayed on top of the original DISP window—if so, you can move the windows so that you can see both at once. If the new windows also have members that are pointers or aggregate types, you can continue to open new DISP windows.

Closing a DISP window

Closing a DISP window is a simple, two-step process.

Step 1: Make the DISP window that you want to close active (see Section 6.4, *The Active Window*, on page xx).

Step 2: Press F4 .

Note that you can close a window and all of its children by closing the original window.

Note:

The debugger automatically closes any DISP windows when you execute a LOAD or SLOAD command.

11.7 Managing Data in a WATCH Window

The debugger doesn't maintain a dedicated window that tells you about the status of all the symbols defined in your program. Such a window might be so large that it wouldn't be useful. Instead, the debugger allows you to create a WATCH window that shows you how program execution affects specific expressions, variables, registers, or memory locations.

watch index —

```
┌─WATCH─────────────────
1:  A0      6
2:  X+X     4
3:  PC      0x00400064
```

label   current value
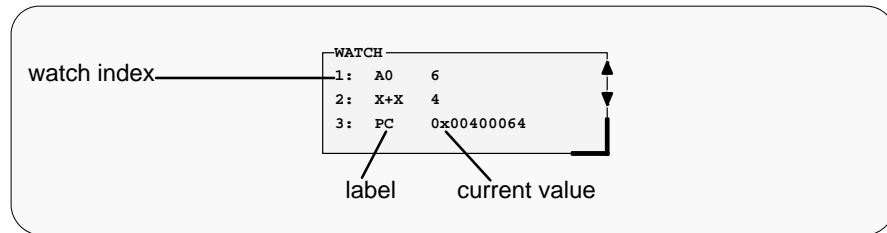
The debugger displays a WATCH window *only when you specifically request a WATCH window* with the WA command (described below). Note that there is only one WATCH window. For additional details concerning the WATCH window, see the *WATCH window* discussion (page xviii).

Remember, you can use the data-management commands or the debugger's overwrite editing capability to modify the contents of any value displayed in the WATCH window. Refer to Section 11.3, *Basic Methods for Changing Data Values* (page vi), for more information.

Note:

All of the watch commands described here can also be accessed from the Watch pulldown menu. For more information about using the the pulldown menus, refer to Section 7.2, *Using the Menu Bar and the Pulldown Menus* (page vii).

```
Watch
Add
Delete
Reset
```
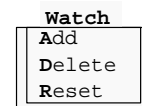
Displaying data in the WATCH window

The debugger has one command for adding items to the WATCH window.

wa   To open the WATCH window, use the WA (watch add) command. The basic syntax is:

wa   *expression* [*, label*]

When you first execute WA, the debugger opens the WATCH window. After that, executing WA adds additional values to the WATCH window.

The *expression* parameter can be any C expression, including an expression that has side effects. It's most useful to watch an expression whose value will change over time; constant expressions provide no useful function in the watch window.

The *label* parameter is optional. When used, it provides a label for the watched entry. If you don't use a *label*, the debugger displays the *expression* in the label field.

Deleting watched values and closing the WATCH window

The debugger supports two commands for deleting items from the WATCH window.

wr   If you'd like to close the WATCH window and delete all of the items in a single step, use the WR (watch reset) command. The syntax is:

wr

wd   If you'd like to delete a specific item from the WATCH window, use the WD (watch delete) command. The syntax is:

wd   *index number*

Whenever you add an item to the WATCH window, the debugger assigns it an index number. (The illustration of the WATCH window on page xx points to these watch indexes.) The WD command's *index number* parameter must correspond to one of the watch indexes in the WATCH window.

Note that deleting an item (depending on where it is in the list) causes the remaining index numbers to be reassigned. Deleting the last remaining item in the WATCH window closes the WATCH window.

Note:

The debugger automatically closes the WATCH window when you execute a LOAD or SLOAD command.

11.8 Displaying Data in Alternative Formats

By default, all data is displayed in its natural format. This means that:

❑ Integer values are displayed as decimal numbers.
❑ Floating-point values are displayed in floating-point format.
❑ Pointers are displayed as hexadecimal addresses (with a 0x prefix).
❑ Enumerated types are displayed symbolically.

However, any data displayed in the COMMAND, MEMORY, WATCH, or DISP window can be displayed in a variety of formats.

Changing the default format for specific data types

To display specific types of data in a different format, use the SETF command. The syntax for this command is:

setf   [*data type*, *display format*]

The *display format* parameter identifies the new display format for any data of type *data type*. Table 2–1 lists the available formats and the corresponding characters that can be used as the *display format* parameter. Table 2–2 lists the C data types that can be used for the *data type* parameter. Only a subset of the display formats applies to each data type, so Table 2–2 also shows valid combinations of data types and display formats.

Table 2–1. Display Formats for Debugger Data

Display Format	Parameter	Display Format	Parameter
Default for the data type	*	Hexadecimal	x
ASCII character (bytes)	c	Octal	o
Decimal	d	Valid address	p
Exponential floating point	e	ASCII string	s
Decimal floating point	f	Unsigned decimal	u

Table 2–2. Data Types for Displaying Debugger Data

Data Type	c	d	o	x	e	f	p	s	u	Default Display Format
char	√	√	√	√					√	ASCII (c)
uchar	√	√	√	√					√	Decimal (d)
short	√	√	√	√					√	Decimal (d)
int	√	√	√	√					√	Decimal (d)
uint	√	√	√	√					√	Decimal (d)

*(Header spanning columns c–u: **Valid Display Formats**)*

Table 2–2.Data Types for Displaying Debugger Data (Continued)

Data Type	**Valid Display Formats**									**Default Display Format**
	c	**d**	**o**	**x**	**e**	**f**	**p**	**s**	**u**	
long	√	√	√	√					√	Decimal (d)
ulong	√	√	√	√					√	Decimal (d)
float			√	√	√	√				Exponential floating point (e)
double			√	√	√	√				Exponential floating point (e)
ptr			√	√			√	√		Address (p)

Here are some examples:

❑ To display all data of type short as an unsigned decimal, enter:

setf short, u ⏎

❑ To return all data of type short to its default display format, enter:

setf short, * ⏎

❑ To list the current display formats for each data type, enter the SETF command with no parameters:

setf ⏎

You'll see a display that looks something like this:

```
        Display Format Defaults
Type char:           ASCII
Type unsigned char:  Decimal
Type int:            Decimal
Type unsigned int:   Decimal
Type short:          Decimal
Type unsigned short: Decimal
Type long:           Decimal
Type unsigned long:  Decimal
Type float:          Exponential floating point
Type double:         Exponential floating point
Type pointer:        Address
```

❑ To reset all data types back to their default display formats, enter :

setf * ⏎

Changing the default format with ?, MEM, DISP, and WA

You can also use the ?, MEM, DISP, and WA commands to show data in alternative display formats. (The ? and DISP commands can use alternative formats only for scalar types, arrays of scalar types, and individual members of aggregate types.)

Each of these commands has an optional *display format* parameter that works in the same way as the *display format* parameter of the SETF command.

When you don't use a *display format* parameter, data is shown in its natural format (unless you have changed the format for the data type with SETF).

Here are some examples:

❑ To watch the PC in decimal, enter:

wa pc,,d ⏎

❑ To display memory contents in octal, enter:

mem 0x0,o ⏎

❑ To display an array of integers as characters, enter:

disp ai,c ⏎

The valid combinations of data types and display formats listed for SETF also apply to the data displayed with DISP, ?, WA, and MEM. For example, if you want to use display format e or f, the data that you are displaying must be of type float or type double. Additionally, you cannot use the s display format parameter with the MEM command.

Using Breakpoints

During the debugging process, you may want to halt execution temporarily so that you can examine the contents of selected variables, registers, and memory locations before continuing with program execution. You can do this by setting **breakpoints** at critical points in your code. You can set breakpoints in assembly language code and in C code. A breakpoint halts any program execution, whether you're running or single-stepping through code.

Breakpoints are especially useful in combination with conditional execution (described on page xv) and benchmarking (emulator only; described on page xvii).

Note that the commands described in this chapter can also be executed from the Break pulldown menu.

Synopsis Page	Topic	
This chapter describes the simple processes of setting and clearing software breakpoints and of obtaining a listing of all the breakpoints that are set.	**12.1 Setting a Breakpoint**	**ii**
	12.2 Clearing a Breakpoint	**iv**
	12.3 Finding the Breakpoints That Are Set	**v**

12.1 Setting a Breakpoint

When you set a breakpoint, the debugger highlights the breakpointed line in two ways:

❏   It prefixes the statement with the characters `BP>`.

❏   It shows the line in a bolder or brighter font. (You can use screen-customization commands to change this highlighting method.)

If you set a breakpoint in the disassembly, the debugger also highlights the associated C statement. If you set a breakpoint in the C source, the debugger also highlights the associated statement in the disassembly. (If more than one assembly language statement is associated with a C statement, the debugger highlights the first of the associated assembly language statements.)

A breakpoint is set at this C statement; notice how the line is highlighted. A breakpoint is also set at the associated assembly language statement (it's highlighted, too).

```
┌FILE: sample.c────────────────────────────────
00057
00058  BP>     meminit();
00059          for (i=0; i < 0x50000;
i++)
00060          {
00061              call(i);
```

```
┌DISASSEMBLY───────────────────────────────────
ffc000a0 0d5f BP>          CALLA  meminit
ffc000d0 a3cf              MOVE   STK,*-SP,1
ffc000e0 932e              MOVE   A9,*STK+,1
ffc000f0 0d5f              CALLA  call
```

Notes:

❏   You cannot set breakpoints in ROM.

❏   After execution is halted by a breakpoint, you can continue program execution by reissuing any of the run or single-step commands.

❏   Up to 200 breakpoints can be set.

There are three ways to set a breakpoint.

↖   1)   Point to the line of assembly language code or C code where you'd like to set a breakpoint.

▌   2)   Click the left button.

Repeating this action clears the breakpoint.

1)   Make the FILE or DISASSEMBLY window the active window.

↑↓   2)   Use the arrow keys to move the cursor to the line of code where you'd like to set a breakpoint.

F9   3)   Press the F9 key.

Repeating this action clears the breakpoint.

ba   If you know the address where you'd like to set a breakpoint, you can use the BA (breakpoint add) command. This command is useful because it doesn't require you to search through code to find the desired line. The syntax for the BA command is:

ba   *address*

This command sets a breakpoint at *address*. This parameter can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label. You cannot set multiple breakpoints at the same statement.

12.2 Clearing a Breakpoint

There are several ways to clear a breakpoint. If you clear a breakpoint from an assembly language statement, the breakpoint is cleared from any associated C statement as well; similiarly, if you clear a breakpoint from a C statement, the breakpoint is cleared from the associated statement in the disassembly.

1) Point to a breakpointed assembly language or C statement.

2) Click the left button.

1) Use the arrow keys or the DASM command to move the cursor to a breakpointed assembly language or C statement.

2) Press the F9 key.

br If you want to clear **all** the breakpoints that are set, use the BR (breakpoint reset) command. The syntax for the BR command is:

br

This command is useful because it doesn't require you to search through code to find the desired line.

bd If you'd like to clear one specific breakpoint and you know the address of this breakpoint, you can use the BD (breakpoint delete) command. The syntax for the BD command is:

bd *address*

This command clears the breakpoint at *address*. This parameter can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label. If no breakpoint is set at *address*, the debugger ignores the command.

12.3 Finding the Breakpoints That Are Set

bl   Sometimes you may need to know where breakpoints are set. For example, the BD command's *address* parameter must correspond to the address of a breakpoint that is set. The BL (breakpoint list) command provides an easy way to get a complete listing of all the breakpoints that are currently set in your program. The syntax for this command is:

bl

The BL command displays a table of breakpoints in the COMMAND window display area. BL lists all the breakpoints that are set, in the order in which you set them. Here's an example of this type of list:

```
 Address    Symbolic Information
ffc000a0    in main, at line 58, "c:\sdb\sample.c"
```

The address is the memory address of the breakpoint. The symbolic information identifies the function, line number, and filename of the breakpointed C statement:

❏ If the breakpoint was set in assembly language code, you'll see only an address unless the statement defines a symbol.

❏ If the breakpoint was set in C code, you'll see the address together with symbolic information.

Customizing the Debugger Display

The debugger display is completely configurable; you can create the interface that is best suited for your use. Besides being able to size and position individual windows, you can change the appearance of many of the display features, such as window borders, how the current statement is highlighted, etc. In addition, if you're using a color display, you can change the colors of any area on the screen. Once you've customized the display to your liking, you can save the custom configuration for use in future debugging sessions.

13.1 Changing the Colors of the Debugger Display

You can use the debugger with a color or a monochrome display; the commands described in this section are most useful if you have a color display. If you are using a monochrome display, these commands change the shades on your display. For example, if you are using a black-and-white display, these commands change the shades of gray that are used.

color
scolor

You can use the COLOR or SCOLOR command to change the colors of areas in the debugger display. The format for these commands is:

color   *area name*, *attribute$_1$* [, *attribute$_2$* [, *attribute$_3$* [, *attribute$_4$*]]]
scolor   *area name*, *attribute$_1$* [, *attribute$_2$* [, *attribute$_3$* [, *attribute$_4$*]]]

These commands are similar. However, SCOLOR updates the screen immediately, and COLOR doesn't update the screen (the new colors/attributes take effect as soon as the debugger executes another command that updates the screen). Typically, you might use the COLOR command several times and follow that with an SCOLOR command to put all of the changes into effect at once.

The *area name* parameter identifies the areas of the display that are affected. The *attributes* identify how the areas are affected. Table 4–1 lists the valid values for the *attribute* parameters.

Table 4–1. Colors and Other Attributes for the COLOR and SCOLOR Commands

(a) Colors

black	blue	green	cyan
red	magenta	yellow	white

(b) Other attributes

bright	blink

The first two *attribute* parameters usually specify the foreground and background colors for the area. If you do not supply a background color, the debugger uses black as the background.

Table 4–2 lists valid values for the *area name* parameters. This is a long list; the subsections following the table further identify these areas.

Table 4–2. Summary of Area Names for the COLOR and SCOLOR Commands

menu_bar	menu_border	menu_entry	menu_cmd
menu_hilite	menu_hicmd	win_border	win_hiborder
win_resize	field_text	field_hilite	field_edit
field_label	field_error	cmd_prompt	cmd_input
cmd_cursor	cmd_echo	asm_data	asm_cdata
asm_label	asm_clabel	background	blanks
error_msg	file_line	file_eof	file_text
file_brk	file_pc	file_pc_brk	

Note: Listing order is left to right, top to bottom.

You don't have to type an entire *attribute* or *area name*; you need to type only enough letters to uniquely identify either parameter. If you supply ambiguous *attribute* names, the debugger interprets the names in this order: black, blue, bright, blink. If you supply ambiguous *area names*, the debugger interprets them in the order that they're listed in Table 4–2 (left to right, top to bottom).

The remainder of this section identifies these areas.

area names: common display areas
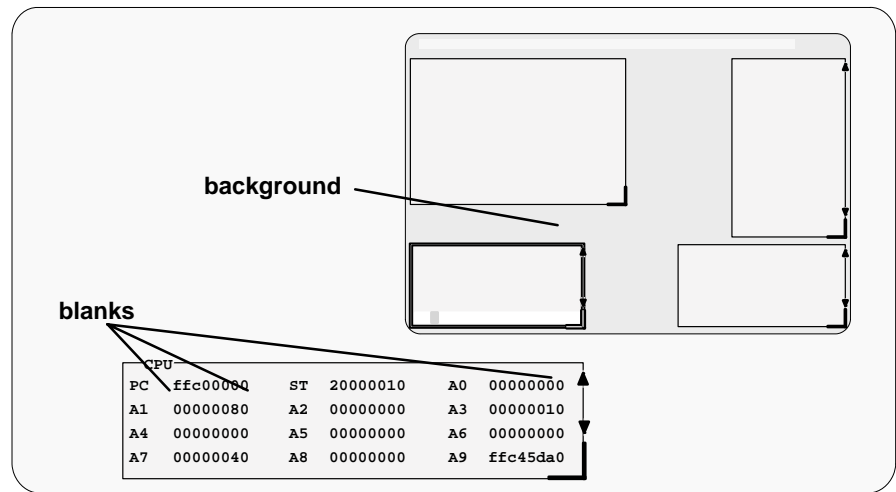
Area identification	Parameter name
Screen background (behind all windows)	background
Window background (inside windows)	blanks

*area names***: window borders**
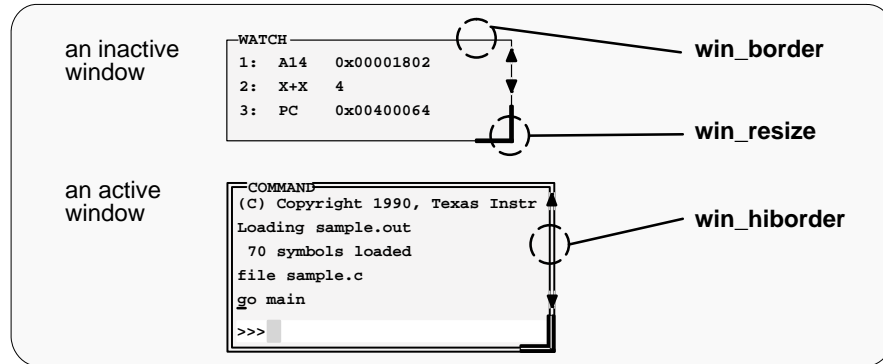
Area identification	Parameter name
Window border for any window that isn't active	win_border
The reversed "L" in the lower right corner of a resizable window	win_resize
Window border of the active window	win_hiborder

*area names***: COMMAND window**
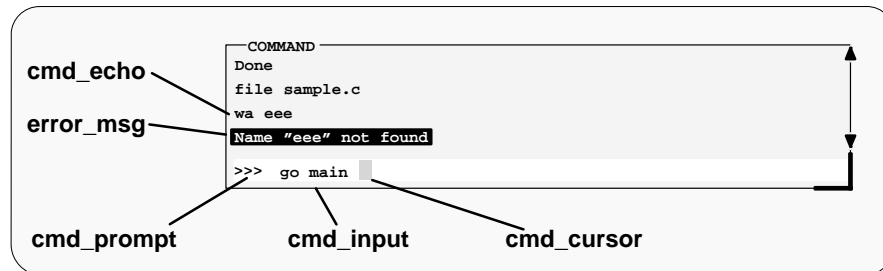
Area identification	Parameter name
Echoed commands in display area	cmd_echo
Errors shown in display area	error_msg
Command-line prompt	cmd_prompt
Text that you enter on the command line	cmd_input
Command-line cursor	cmd_cursor

*area names***: DISASSEMBLY and FILE windows**

asm_cdata

```
┌DISASSEMBLY───────────────────────────────────────┐
│ffc00000   098f    main:     MMTM   SP,A7,A9,A11,F │ ──→ file_pc
│P                                                  │
│ffc00020   4dcd              MOVE   STK,FP          │
│ffc00030   0b0e              ADDI   64,STK          │
│ffc00050   93ee              MOVE   SP,*STK+,1      │ ──→ file_brk
│ffc00060   4529              SUB    A9,A9           │
└───────────────────────────────────────────────────┘
```

asm_clabel
asm_label

asm_data

file_text

file_line

```
┌FILE: t1.c ─────────────────────────┐
│00053   extern    call();           │
│00054   extern    meminit():        │
│00055   main()                      │ ──→ file_pc_brk
│00056   {                           │
│*eof                                │
└─────────────────────────────────────┘
```

file_eof
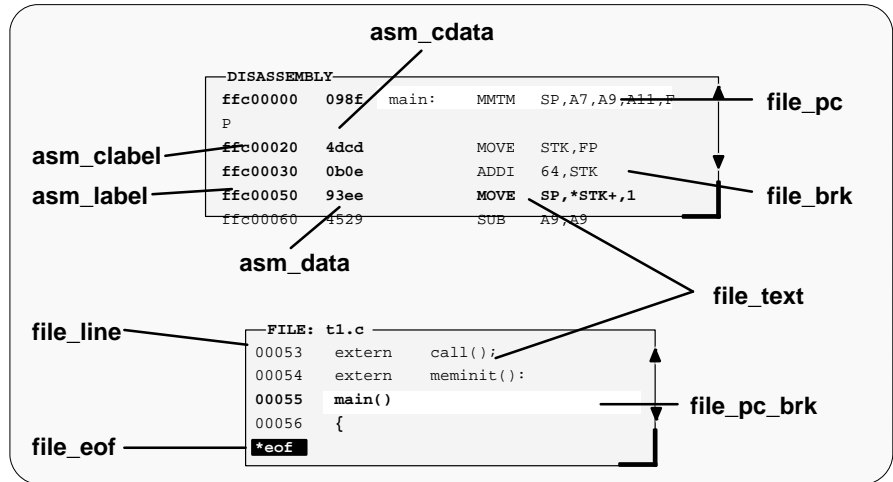
Area identification	Parameter name
Object code in DISASSEMBLY window that is associated with current C statement	asm_cdata
Object code in DISASSEMBLY window	asm_data
Addresses in DISASSEMBLY window	asm_label
Addresses in DISASSEMBLY window that are associated with current C statement	asm_clabel
Line numbers in FILE window	file_line
End-of-file marker in FILE window	file_eof
Text in FILE or DISASSEMBLY window	file_text
Breakpointed text in FILE or DISASSEMBLY window	file_brk
Current PC in FILE or DISASSEMBLY window	file_pc
Breakpoint at current PC in FILE or DISASSEMBLY window	file_pc_brk

area names: **data-display windows**
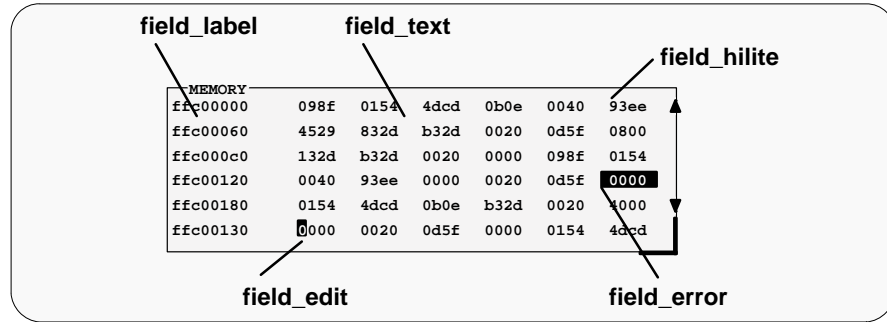
Area identification	Parameter name
Label of a window field (includes register names in CPU window, addresses in MEMORY window, index numbers and labels in WATCH window, member names in DISP window)	field_label
Text of a window field (includes data values for all data-display windows) and of most command output messages in command window	field_text
Text of a highlighted field	field_hilite
Text of a field that has an error (such as an invalid memory location)	field_error
Text of a field being edited (includes data values for all data-display windows)	field_edit

*area names***: menu bar and pulldown menus**
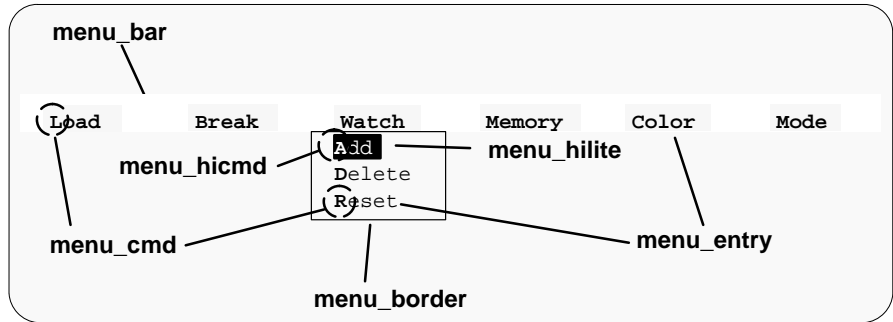
Area identification	Parameter name
Top line of display screen; background to main menu choices	menu_bar
Border of any pulldown menu	menu_border
Text of a menu entry	menu_entry
Invocation key for a menu or menu entry	menu_cmd
Text for current (selected) menu entry	menu_hilite
Invocation key for current (selected) menu entry	menu_hicmd

13.2 Changing the Border Styles of the Windows

In addition to changing the colors of areas in the display, the debugger allows you to modify the border styles of the windows.

border Use the BORDER command to change window border styles. The format for this command is:

border [*active window style*] [, *inactive window style*] [, *resize style*]

This command can change the border styles of the active window, the inactive windows, and any window that is being resized. The debugger supports nine border styles. Each parameter for the BORDER command must be one of the numbers that identifies these styles:

Index	Style
0	Double-lined box
1	Single-lined box
2	Solid 1/2-tone top, double-lined sides and bottom
3	Solid 1/4-tone top, double-lined sides and bottom
4	Solid box, thin border
5	Solid box, heavy sides, thin top and bottom
6	Solid box, heavy borders
7	Solid 1/2-tone box
8	Solid 1/4-tone box

Here are some examples of the BORDER command. If desired, you can skip parameters as shown in the examples.

```
border 6,7,8          Change style of active, inactive, and resize windows
border 1,,2               Change style of active and resize windows
border ,3                            Change style of inactive window
```

You can execute the BORDER command as the Border selection on the Color pulldown menu. The debugger displays a dialog box so that you can enter the parameter values; in the dialog box, *active window style* is called *foreground*, and *inactive window style* is called *background*.

13.3 Saving and Using Custom Displays

The debugger allows you to save and use as many custom configurations as you like.

When you invoke the debugger, it looks for a screen configuration file named init.clr. This file defines how various areas of the display will appear. If the debugger doesn't find init.clr, it uses a default screen configuration. Initially, init.clr defines a screen configuration that exactly matches the default configuration.

The debugger supports two commands for saving and restoring custom screen configurations into files. The filenames that you use for restoring configurations must correspond to the filenames that you used for saving configurations. Note that these are binary files, not text files, so you can't edit the files with a text editor.

Changing the default display for monochrome monitors

The default display is most useful with color monitors. The debugger highlights changed values, messages, and other information with color; this may not be particularly helpful if you are using a monochrome monitor.

The debugger package includes another screen configuration file named mono.clr that defines a screen configuration that can be used with monochrome monitors. The best way to use this configuration is to rename the file:

1) Rename the original init.clr file—you might want to call it color.clr.

2) Now rename the mono.clr file. Call it init.clr. Now, whenever you invoke the debugger, it will automatically come up with a customized screen configuration for monochrome files.

If you aren't happy with the way that this file defines the screen configuration, you can customize it.

Saving a custom display

ssave   Once you've customized the debugger display to your liking, you can use the SSAVE command to save the current screen configuration to a file. The format for this command is:

ssave   [*filename*]

This saves the screen colors, window positions, window sizes, and border styles for all debugging modes. The *filename* parameter names the new screen configuration file. You can include path information (including relative pathnames); if you don't supply path information, the debugger places the file in the current directory.

If you don't supply a *filename*, then the debugger saves the current configuration into a file named init.clr in the current directory.

You can execute this command as the Save selection on the Color pulldown menu.

Loading a custom display

sconfig   You can use the SCONFIG command to restore the display to a particular configuration. The format for this command is:

sconfig   [*filename*]

This restores the screen colors, window positions, window sizes, and border styles that were saved with the SSAVE command into *filename*. If you don't supply a *filename*, the debugger looks for the init.clr file. The debugger searches for the specified file (or for init.clr) in the current directory and then in directories named with the D_DIR environment variable.

You can execute this command as the Load selection on the Color pulldown menu.

Invoking the debugger with a custom display

If you set up the screen in a way that you like and always want to invoke the debugger with this screen configuration, you have two choices for accomplishing this:

❏ Save the configuration in init.clr.

❏ Add a line to the init.cmd file (the batch file that the debugger executes at invocation time); this line should use the SCONFIG command to load the custom configuration.

Returning to the default display

If you saved a custom configuration into init.clr but don't want the debugger to come up in that configuration, then rename the file or delete it. If you are in the debugger, have changed the configuration, and would like to revert to the default, just execute the SCONFIG command without a filename.

13.4 Changing the Prompt

prompt The debugger enables you to change the command-line prompt by using the PROMPT command. The format of this command is:

prompt *new prompt*

The *new prompt* can be any string of characters, excluding semicolons and commas. (If you type a semicolon or a comma, it terminates the prompt string.)

The SSAVE command doesn't save the command-line prompt as part of a custom configuration. The SCONFIG command doesn't change the command-line prompt. If you change the prompt, it stays changed until you change it again, even if you use SCONFIG to load a different screen configuration.

If you always want to use a different prompt, you can add a PROMPT statement to the init.cmd file (the batch file that the debugger executes at invocation time).

You can also execute this command as the Prompt selection on the Color pulldown menu.

Summary of Commands and Special Keys

This chapter summarizes the debugger's commands and special key sequences.

14.1 Functional Summary of Debugger Commands

This section summarizes the debugger commands according to these categories:

❏ **Changing modes.** These commands enable you to switch freely between the three debugging modes (auto, mixed, and assembly). You can select these commands from the Mode pulldown menu, also.

❏ **Managing windows.** These commands enable you to select the active window and move or resize the active window. You can perform these functions with the mouse, also.

❏ **Performing system tasks.** These commands enable you to perform several DOS-like functions and provide you with some control over the target system.

❏ **Displaying and changing data.** These commands enable you to display and evaluate a variety of data items. Some of these commands are available on the Watch pulldown menu, also.

❏ **Displaying files and loading programs.** These commands enable you to change the displays in the FILE and DISASSEMBLY windows and to load object files into memory. Several of these commands are available on the Load pulldown menu.

❏ **Managing breakpoints.** These commands provide you with a command-line method for controlling software breakpoints. These commands are available through the Break pulldown menu. You can also set/clear breakpoints interactively.

❏ **Customizing the screen.** These commands allow you to customize the debugger display, then save and later reuse the customized displays. These commands are available from the Color pulldown menu, also.

❏ **Memory mapping.** These commands enable you to define the areas of target memory that the debugger can access or to fill a memory range with an initial value. These commands are available on the Memory pulldown menu, also.

❏ **Running programs.** These commands provide you with a variety of methods for running your programs in the debugger environment. The basic run and single-step commands are available on the menu bar, also.

Changing modes

To do this	Use this command	See page
Put the debugger in assembly mode	asm	ix
Put the debugger in auto mode for debugging C code	c	xi
Put the debugger in mixed mode	mix	xxi

Managing windows

To do this	Use this command	See page
Make the active window as large as possible	zoom	xxxvii
Select the active window	win	xxxvi
Reposition the active window	move	xxii
Resize the active window	size	xxx

Performing system tasks

To do this	Use this command	See page
Define your own command string	alias	viii
Associate a beeping sound with the display of error messages	sound	xxxi
Enter any operating-system command or exit to a system shell	system	xxxii
Delete an alias definition	unalias	xxxiv
Clear all displayed information from the COMMAND window display area	cls	xii
Change the current working directory from within the debugger environment	cd/chdir	xii
List the contents of the current directory or any other directory	dir	xv
Name additional directories that can be searched when you load source files	use	xxxv
Execute commands from a batch file	take	xxxiii
Reset the target system (emulator only) or reload gspmon.out (development boards only)	reset	xxv
Exit the debugger	quit	xxiv

Displaying and changing data

To do this	Use this command	See page
Change the default format for displaying data values	setf	xxix
Show the type of a data item	whatis	xxxvi
Evaluate and display the result of a C expression	?	vii
Evaluate a C expression without displaying the results	eval	xvi
Display the values in an array or structure or display the value that a pointer is pointing to	disp	xv
Display a different range of memory in the MEMORY window	mem	xx
Open the FPU window	fpuregs	xvii
Open the I/O window	ioregs	xviii
Continuously display the value of a variable, register, or memory location within the WATCH window	wa	xxxv
Delete a data item from the WATCH window	wd	xxxvi
Delete all data items from the WATCH window and close the WATCH window	wr	xxxvii

Displaying files and loading programs

To do this	Use this command	See page
Display a text file in the FILE window	file	xvi
Display C and/or assembly language code at a specific point	addr	viii
Display assembly language code at a specific address	dasm	xiv
Display a specific C function	func	xvii
Reopen the CALLS window	calls	xi
Load an object file	load	xix
Load only the object-code portion of an object file	reload	xxiv
Load only the symbol-table portion of an object file	sload	xxxi

Managing breakpoints

To do this	Use this command	See page
Add a breakpoint	ba	ix
Delete a breakpoint	bd	ix
Display a list of all the breakpoints that are set	bl	x
Reset (delete) all breakpoints	br	xi

Loading TIGA applications

To do this	Use this command	See page
Load a previously loaded TIGA module or list the modules that were dynamically loaded	mod	xxi
Set a tentative breakpoint	tba	xxxiv
Clear a tentative breakpoint	tbd	xxxiv

Customizing the screen

To do this	Use this command	See page
Change the screen colors and update the screen immediately	scolor	xxvii
Change the screen colors, but don't update the screen immediately	color	xiii
Change the border style of any window	border	x
Change the command-line prompt	prompt	xxiv
Save a custom screen configuration	ssave	xxxi
Load and use a previously saved custom screen configuration	sconfig	xxviii

Memory mapping

To do this	Use this command	See page
Initialize a block of memory	fill	xvii
Save a block of memory to a system file	ms	xxiii
Enable or disable memory mapping	map	xx
Add an address range to the memory map	ma	xix
Delete an address range from the memory map	md	xx
Reset (delete all ranges) the memory map	mr	xxiii
Display a list of the current memory map settings	ml	xxi

Running programs

To do this	Use this command	See page
Run a program	run	xxvi
Run a program up to a certain point	go	xviii
Single-step through assembly language or C code	step	xxxii
Single-step through assembly language or C code, one C statement at a time	cstep	xiv
Single-step through assembly language or C code; step over function calls	next	xxiii
Single-step through assembly language or C code one C statement at a time; step over function calls	cnext	xii
Run a program with benchmarking—count the number of CPU clock cycles consumed by the executing portion of code (emulator only)	runb	xxvi
Execute code in a function and return to the function's caller	return	xxv
Reset the program entry point	restart	xxv
Disconnect the emulator from the target system and run free (emulator only)	runf	xxvii
Halt the target system after executing a RUNF command (emulator only)	halt	xviii
Execute commands from a batch file	take	xxxiii
Reset the target system (emulator only) or reload gspmon.out (development boards only)	reset	xxv

14.2 Alphabetical Summary of Debugger Commands

Commands are not case sensitive; to emphasize this, command names are shown in both uppercase and lowercase throughout this book.

?   *Evaluate Expression*

Syntax   **?**   *expression* [*, display format*]

Menu selection   none

Description   The ? (evaluate expression) command evaluates an expression and shows the result in the COMMAND window display area. The *expression* can be any C expression, including an expression with side effects; however, you cannot use a string constant or function call in the *expression.*

If the result of *expression* is not an array or structure, then the debugger displays the results in the COMMAND window. If *expression* is a structure or array, ? displays the entire contents of the structure or array; you can halt long listings by pressing ⌨ESC .

When you use the optional *display format* parameter, data will be displayed in one of the following formats:

Parameter	Result	Parameter	Result
*	Default for the data type	x	Hexadecimal
c	ASCII character (bytes)	o	Octal
d	Decimal	p	Valid address
e	Exponential floating point	s	ASCII string
f	Decimal floating point	u	Unsigned decimal

addr	*Display Code at Selected Address*

Syntax

    addr   *address*
    addr   *function name*

Menu selection   none

Description   Use the ADDR command to display C code or the disassembly at a specific point. ADDR's behavior changes, depending on the current debugging mode:

❑ In assembly mode, ADDR works like the DASM command, positioning the code starting at *address* or at *function name* as the first line of code in the DISASSEMBLY window.

❑ In a C display, ADDR works like the FUNC command, displaying the code starting at *address* or at *function name* in the FILE window.

❑ In mixed mode, ADDR affects both the DISASSEMBLY and FILE windows.

The *address* parameter is treated as a program-memory address.

Note:

ADDR affects the FILE window only if the specified *address* is in a C function.

---

alias	*Define Custom Command String*

Syntax   **alias**   [*alias name* [, "*command string*"]]

Menu selection   none

Description   The ALIAS command allows you to associate one or more debugger commands with a single *alias name*. You can include as many debugger commands in the *command string* as you like, as long you separate them with semicolons and enclose the entire string of commands in quotation marks. You can also identify debugger-command parameters by a percent sign followed by a number (%1, %2, etc.). The total number of characters for an individual command (expanded to include parameter values) is limited to 132.

Previously defined alias names can be included as part of the definition for a new alias.

To find the current definition of an alias, enter the ALIAS command with the *alias name* only. To see a list of all defined aliases, enter the ALIAS command with no parameters.

asm	*Enter Assembly-Only Debugging Mode*

Syntax   **asm**

Menu selection   Mo**D**e→**A**sm

Description   The ASM command changes from the current debugging mode to assembly mode. If you're already in assembly mode, the ASM command has no effect.

ba	*Breakpoint Add*

Syntax   **ba** *address*

Menu selection   **B**reak→**A**dd

Description   The BA command sets a breakpoint at a specific *address*. This command is useful because it doesn't require you to search through code to find the desired line. The *address* can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label.

bd	*Breakpoint Delete*

Syntax   **bd** *address*

Menu selection   **B**reak→ **D**elete

Description   The BD command clears a breakpoint at a specific *address*. The *address* can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label.

bl	*Breakpoint List*

Syntax   **bl**

Menu selection   **B**reak→**L**ist

Description   The BL command provides an easy way to get a complete listing of all the breakpoints that are currently set in your program. It displays a table of breakpoints in the COMMAND window display area. BL lists all the breakpoints that are set, in the order in which you set them.

border	*Change Style of Window Border*

Syntax   **border**   [*active window style*] [[*,inactive window style*] [*,resize window style*]

Menu selection   **C**olor→**B**order

Description   The BORDER command changes the border style of the active window, the inactive windows, and any window that you're resizing. The debugger supports nine border styles. Each parameter for the BORDER command must be one of the numbers that identifies these styles:

Index	Style
0	Double-lined box
1	Single-lined box
2	Solid 1/2-tone top, double-lined sides/bottom
3	Solid 1/4-tone top, double-lined sides/bottom
4	Solid box, thin border
5	Solid box, heavy sides, thin top/bottom
6	Solid box, heavy borders
7	Solid 1/2-tone box
8	Solid 1/4-tone box

Note that you can execute the BORDER command as the Border selection on the Color pulldown menu. The debugger displays a dialog box so that you can enter the parameter values; in the dialog box, *active window style* is called *foreground*, and *inactive window style* is called *background*.

br	*Breakpoint Reset*

Syntax   **br**

Menu selection   **B**reak→**R**eset

Description   The BR command clears all breakpoints that are set.

c	*Enter Auto Debugging Mode*

Syntax   **c**

Menu selection   Mo**D**e→**C** (auto)

Description   The C command changes from the current debugging mode to auto mode. If you're already in auto mode, then the C command has no effect.

calls	*Open CALLS Window*

Syntax   **calls**

Menu selection   none

Description   The CALLS command displays the CALLS window. The debugger displays this window automatically when you are in auto/C or mixed mode. However, you can close the CALLS window; the CALLS command opens the window up again.

cd, chdir	*Change Directory*

Syntax	**cd** [*directory name*]
	chdir [*directory name*]
Menu selection	none
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 you don't use a *pathname*, 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. You can also use the CD command to change the current drive. For example:

```
cd a:
cd d:\csource
cd c:\emu34020
```

cls	*Clear Screen*

Syntax	**cls**
Menu selection	none
Description	The CLS command clears all displayed information from the COMMAND window display area.

cnext	*Single-Step C, Next Statement*

Syntax	**cnext** [*expression*]
Menu selection	Next=**F10** (in C code)
Description	The CNEXT command is similar to the CSTEP command. It runs a program one C statement at a time, updating the display after executing each statement. If you're using CNEXT to step through assembly language code, the debugger won't update the display until it has executed all assembly language statements associated with a single C statement. Unlike CSTEP, CNEXT steps over function calls rather than stepping into them—you don't see the single-step execution of the function call.

The *expression* parameter specifies the number statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page xv, discusses this in detail).

| **color** | *Change Screen Colors* |

Syntax   **color**   *area name, attribute₁* [*,attribute₂* [*,attribute₃* [*,attribute₄*]]]

Menu selection   none

Description   The COLOR command changes the color of specified areas of the debugger display. COLOR doesn't update the display; the changes take effect when another command, such as SCOLOR, updates the display. The *area name* parameter identifies the area of the display that is affected. The *attributes* identify how the area is affected. The first two *attribute* parameters usually specify the foreground and background colors for the area. If you do not supply a background color, the debugger uses black as the background.

Valid values for the *attribute* parameters include:

black	blue	green	cyan
red	magenta	yellow	white
bright		blink	

Valid values for the *area name* parameters include:

menu_bar	menu_border	menu_entry	menu_cmd
menu_hilite	menu_hicmd	win_border	win_hiborder
win_resize	field_text	field_hilite	field_edit
field_label	field_error	cmd_prompt	cmd_input
cmd_cursor	cmd_echo	asm_data	asm_cdata
asm_label	asm_clabel	background	blanks
error_msg	file_line	file_eof	file_text
file_brk	file_pc	file_pc_brk	

You don't have to type an entire *attribute* or *area name*; you need to type only enough letters to uniquely identify the attribute. If you supply ambiguous *attribute* names, the debugger interprets the names in this order: black, blue, bright, blink. If you supply ambiguous *area names*, the debugger interprets them in the order that they're listed above (left to right, top to bottom).

cstep   *Single-Step C*

Syntax   **cstep**   [*expression*]

Menu selection   Step=**F8** (in C code)

Description   The CSTEP single-steps through a program, one C statement at a time, updating the display after executing each statement. If you're using CSTEP to step through assembly language code, the debugger won't update the display until it has executed all assembly language statements associated with a single C statement.

If you're single-stepping through C code and encounter a function call, the STEP command shows you the single-step execution of the called function (assuming that the function was compiled with the compiler's –g debug option). When function execution completes, single-step execution returns to the caller. If the function wasn't compiled with the debug option, the debugger executes the function but doesn't show single-step execution of the function.

The *expression* parameter specifies the number statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page xv, discusses this in detail).

dasm   *Display Selected Disassembly*

Syntax   **dasm** *address*
                    dasm *function name*

Menu selection   none

Description   The DASM command displays code beginning at a specific point within the DISASSEMBLY window. The *address* parameter is treated as a program-memory address.

dir	*Show Directory Contents*

Syntax **dir** [*directory name*]

Menu selection none

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 the parameter, the debugger lists the contents of the current directory.

disp	*Open DISPlay Window*

Syntax **disp** *expression* [*, display format*]

Menu selection none

Description The DISP command opens a DISP window to display the contents of an array, structure, or pointer expressions to a scalar type (of the form **pointer*). If the *expression* is not one of these types, then DISP acts like a ? command. You can have up to 120 DISP windows open at the same time.

Once you open a DISP window, you may find that a displayed member is itself an array, structure, or pointer:

A member that is an array looks like this   [. . .]
A member that is a structure looks like this   {. . .}
A member that is a pointer looks like an address   0x00000000

You can display the additional data (the data pointed to or the members of the array or structure) in another DISP window by using the DISP command again, using the arrow keys to select the field and then pressing F9, or pointing the mouse cursor to the field and pressing the left mouse button.

When you use the optional *display format* parameter, data will be displayed in one of the following formats:

Parameter	Result	Parameter	Result
*	Default for the data type	**x**	Hexadecimal
c	ASCII character (bytes)	**o**	Octal
d	Decimal	**p**	Valid address
e	Exponential floating point	**s**	ASCII string
f	Decimal floating point	**u**	Unsigned decimal

The *display format* parameter can be used only when you are displaying a scalar type, an array of scalar type, or a individual member of an aggregate type.

You can also use the DISP command with a typecast expression to display memory contents in any format. Here are some examples:

```
disp *0
disp *(float *)123
disp *(char *)0x111
```

This shows memory in the DISP window as an array of locations; the location that you specify with the *expression* parameter is member [0], and all other locations are offset from that location.

eval	*Evaluate Expression*

Syntax	**eval** *expression* **e** *expression*
Menu selection	none
Description	The EVAL command evaluates an expression like the ? command does *but does not show the result* in the COMMAND window display area. EVAL is useful for assigning values to registers or memory locations in a batch file (where it's not necessary to display the result).

file	*Display Text File*

Syntax	**file** *filename*
Menu selection	**L**oad→**F**ile
Description	The FILE command displays the contents of any text file in the FILE window. The debugger continues to display this file until you run a program and halt in a C function. This command is intended primarily for displaying C source code. You can view only one text file at a time. You are restricted to displaying files that are 65,518 bytes long or less.

fill	*Fill Memory*

Syntax   **fill** *address, length,data*

Menu selection   **M**emory→**F**ill

Description   The FILL command fills a block of memory with a specified value. This command has three parameters:

❑ The *address* parameter identifies the beginning of the block.
❑ The *length* parameter defines the number of 32-bit words that will be filled.
❑ The *data* is the value that the memory block will be filled with.

fpuregs	*Display '34082 Registers*

Syntax   **fpuregs**

Menu selection   none

Environments   ☑ basic debugger   ☐ profiler

Description   The FPUREGS command opens the FPU window to display '34082 registers. The window shows '34082 A- and B-file registers in double-precision scientific notation and status/configuration registers as hexadecimal values.

Note:

The FPU window can be displayed only if you've invoked the debugger with the –mc option.

---

func	*Display Function*

Syntax   **func** *function name*
                     func *address*

Menu selection   none

Description   The FUNC command displays a specified C function in the FILE window. You can identify the function by its name or its address; an *address* parameter is treated as a program-memory address. FUNC works similarly to FILE, but when you use FUNC, you don't need to identify the name of the file that contains the function.

go   *Run to Specified Address*

Syntax   **go**   [*address*]

Menu selection   none

Description   The GO command executes code up to a specific point in your program. The *address* parameter is treated as program-memory address. If you don't supply an *address*, then GO acts like a RUN command without an *expression* parameter.

halt   *Halt Target System*

Syntax   **halt**

Menu selection   none

Description   The HALT command halts the target system after you've entered a RUNF command. When you invoke the debugger, it automatically executes a HALT command. Thus, if you enter a RUNF, quit the debugger, and later reinvoke the debugger, you will effectively reconnect the emulator to the target system and run the debugger in its normal mode of operation.

Note:

This command is for the emulator only; it does not work with the development board version of the debugger. If you attempt to use the HALT command with a development board, the debugger displays this error message:

```
--- Execution error
```

---

ioregs   *Display I/O registers*

Syntax   **ioregs**

Menu selection   none

Environments   ☑   basic debugger   ☐   profiler

Description   The IOREGS command opens the I/O window to display the '340 I/O registers.

load	*Load Object File*

Syntax	**load** *object filename*
Menu selection	**L**oad→ **L**oad
Description	The LOAD command loads both an object file and its associated symbol table into memory. In effect, the LOAD command performs both a RELOAD and an SLOAD. Note that the LOAD command clears the old symbol table and closes the WATCH and DISP windows.
	Do not use the LOAD command while debugging TIGA applications.

ma	*Memory Map Add*

Syntax	**ma** *address, length, type*
Menu selection	**M**emory→**A**dd
Description	The MA command identifies valid ranges of target memory. This command has three parameters:

❏ The *address* parameter defines the starting address of a range of memory. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.

❏ The *length* parameter defines the length of the range. This parameter can be any C expression.

❏ The *type* parameter identifies the read/write characteristics of the memory range. The *type* must be one of these keywords:

To identify this kind of memory	**Use this keyword as the *type* parameter**
read-only memory	**R**, **ROM**, or **READONLY**
write-only memory	**W**, **WOM**, or **WRITEONLY**
read/write memory	**RW** or **RAM**
no-access memory	**PROTECT**

A new memory map must not overlap an existing entry. If you define a range that overlaps an existing range, the debugger ignores the new range.

map　　　　　　　　　*Enable Memory Mapping*

Syntax　　　　　　　　**map**　{**on** | **off**}

Menu selection　　　　**M**emory→**E**nable

Description　　　　　　The MAP command enables or disables memory mapping. In some instances, you may want to explicitly enable or disable memory. Note that disabling memory mapping can cause bus fault problems in the target because the debugger may attempt to access nonexistent memory.

md　　　　　　　　　*Memory Map Delete*

Syntax　　　　　　　　**md**　*address*

Menu selection　　　　**M**emory→**D**elete

Description　　　　　　The MD command deletes a range of memory from the debugger's memory map. The *address* parameter identifies the starting address of a range of memory.

mem　　　　　　　　*Modify MEMORY Window Display*

Syntax　　　　　　　　**mem**　*expression* [*, display format*]

Menu selection　　　　none

Description　　　　　　The MEM command identifies a new starting address for the block of memory displayed in the MEMORY window. The debugger displays the contents of memory at *expression* in the first data position in the MEMORY window. The end of the range is defined by the size of the window. The *expression* can be an absolute address, a symbolic address, or any C expression.

When you use the optional *display format* parameter, memory will be displayed in one of the following formats:

Parameter	Result	Parameter	Result
*	Default for the data type	**x**	Hexadecimal
c	ASCII character (bytes)	**o**	Octal
d	Decimal	**p**	Valid address
e	Exponential floating point	**u**	Unsigned decimal
f	Decimal floating point		

mix — *Enter Mixed Debugger Mode*

Syntax   **mix**

Menu selection   Mo**D**e→**M**ixed

Description   The MIX command changes from the current debugging mode to mixed mode. If you're already in mixed mode, the MIX command has no effect.

ml — *Memory Map List*

Syntax   **ml**

Menu selection   **M**emory→**L**ist

Description   The ML command lists the memory ranges that are defined for the debugger's memory map. The ML command lists the starting address, ending address, and read/write characteristics of each defined memory range.

mod — *TIGA Module Identifier*

Syntax   **mod** [*TIGA module name*]

Menu selection   none

Description   Once a module has been selected and dynamically loaded, it can be selected with the MOD command. The newly selected module becomes the default module, which helps avoid misinterpretation of variable names used in multiple modules.

If you would like a list of the modules that have been dynamically loaded, enter the MOD command with no parameters.

move	*Move Active Window*

Syntax **move** [*X position*, *Y position* [, *width*, *length*]]

Menu selection none

Description The MOVE command moves the upper left corner of the active window to the specified XY position. If you choose, you can resize the window while you move it (see the SIZE command for valid *width* and *length* values). You can use the MOVE command in one of two ways:

❑ By supplying a specific *X position* and *Y position* or
❑ By omitting the *X position* and *Y position* parameters and using function keys to interactively move the window.

Valid X and Y positions depend on the screen size and the window size. These are the minimum and maximum XY positions. The maximum values assume that the window is as small as possible; for example, if a window was half as tall as the screen, you wouldn't be able to move its upper left corner to an X position on the bottom half of the screen.

Screen size	Debugger options	Valid X positions	Valid Y positions
80 characters by 25 lines	none	0 through 76	1 through 22
80 characters by 43 lines (EGA) 80 characters by 50 lines (VGA)	–b	0 through 76	1 through 40 1 through 47
120 characters by 43 lines	–bb	0 through 116	1 through 40
132 characters by 43 lines	–bbb	0 through 128	1 through 40
80 characters by 60 lines	–bbbb	0 through 76	1 through 57
100 characters by 60 lines	–bbbbb	0 through 106	1 through 57

Note: To use larger screen sizes, you must invoke the debugger with the appropriate –b option.

If you enter the MOVE command without *X position* and *Y position* parameters, you can use arrow keys to move the window:

⬇   Moves the active window down one line.
⬆   Moves the active window up one line.
⬅   Moves the active window left one character position.
➡   Moves the active window right one character position.

When you're finished using the arrow keys, you *must* press ESC or ⤶.

---

mr	*Memory Map Reset*

Syntax   **mr**

Menu selection   **M**emory→**R**eset

Description   The MR command resets the debugger's memory map by deleting all defined
                     memory ranges from the map.

ms	*Save a Block of Memory to a File*

Syntax   **ms** *address, length,filename*

Menu selection   **M**emory→**S**ave

Description   The MS command saves the values in a block of memory to a system file. The
                     command has three parameters:

❑ The *address* parameter identifies the beginning of the block.
❑ The *length* parameter defines the length, in words, of the block.
❑ The *filename* is a system file. Files are saved in COFF format.

next	*Single-Step, Next Statement*

Syntax   **next**   [*expression*]

Menu selection   Next=**F10** (in disassembly)

Description   The NEXT command is similar to the STEP command. If you're in C code, the
                     debugger executes one C statement at a time. In assembly or mixed mode,
                     the debugger executes one assembly language statement at a time. Unlike
                     STEP, NEXT never updates the display when executing called functions;
                     NEXT always steps to the next consecutive statement. Unlike STEP, NEXT
                     steps over function calls rather than stepping into them—you don't see the
                     single-step execution of the function call.

The *expression* parameter specifies the number statements that you want to
single-step. You can also use a conditional *expression* for conditional
single-step execution (*Running code conditionally*, page xv, discusses this in
detail).

prompt	*Change Command-Line Prompt*

Syntax　　　　　　　**prompt**　*new prompt*

Menu selection　　　**C**olor→**P**rompt

Description　　　　　The PROMPT command changes the command-line prompt. The *new prompt* can be any string of characters (note that a semicolon or comma ends the string).

quit	*Exit Debugger*

Syntax　　　　　　　**quit**

Menu selection　　　none

Description　　　　　The QUIT command exits the debugger and returns to the DOS environment.

reload	*Reload Object Code*

Syntax　　　　　　　**reload**　*object filename*

Menu selection　　　**L**oad→**R**eload

Description　　　　　The RELOAD command loads only an object file *without* loading its associated symbol table. This is useful for reloading a program when target memory has been corrupted.

　　　　　　　　　　　Do not use the RELOAD command while debugging TIGA applications.

reset	*Reset Target System*

Syntax   **reset**

Menu selection   **L**oad→Rese**T**

Description   The RESET command works differently for the various '340 debugging systems:

❏ For the emulator, RESET resets the target system. This is a *software* reset.

❏ For development boards, RESET reloads the monitor (gspmon) but does not reset the '340 device.

restart	*Reset PC to Program Entry Point*

Syntax   **restart**
                      rest

Menu selection   **L**oad→R**E**start

Description   The RESTART or REST command resets the program to its entry point. (This assumes that you have already used one of the load commands to load a program into memory.)

return	*Return to Function's Caller*

Syntax   **return**
                      ret

Menu selection   none

Description   The RETURN or RET command executes the code in the current C function and halts when execution reaches the caller. Breakpoints do not affect this command, but you can halt execution by pressing the left mouse button or pressing ESC .

---

run	*Run Code*

Syntax   **run**   [*expression*]

Menu selection   Run=**F5**

Description   The RUN command is the basic command for running an entire program. The command's behavior depends on the type of parameter you supply:

❑ If you don't supply an *expression*, the program executes until it encounters a breakpoint or until you press the left mouse button or press ESC .

❑ If you supply a logical or relational *expression*, this becomes a conditional run (described in detail on page xv).

❑ If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger executes *count* instructions, halts, and updates the display.

---

runb	*Run Benchmark*

Syntax   **runb**

Menu selection   none

Description   The RUNB command executes a specific section of code and counts the number of CPU clock cycles consumed by the execution. In order to operate correctly, *execution must be halted by a breakpoint*. After RUNB execution halts, the debugger stores the number of cycles into the CLK pseudoregister. For a complete explanation of the RUNB command and the benchmarking process, read Section 9.7, *Benchmarking*, on page xvii.

Note: Emulator Only

This command is for the emulator only; it does not work with the development board version of the debugger. If you attempt to use the RUNB command with a development board, the debugger displays this error message:

```
                        --- Execution error
```

---

runf	*Run Free*

Syntax   **runf**

Menu selection   none

Description   The RUNF command disconnects the emulator from the target system while code is executing. When you enter RUNF, the debugger clears all breakpoints, disconnects the emulator from the target system, and causes the processor to begin execution at the current PC. You can quit the debugger, or you can continue to enter commands. However, any command that causes the debugger to access the target at this time produces an error.

The HALT command stops a RUNF; the debugger automatically executes a HALT when the debugger is invoked.

Note: Emulator Only

This command is for the emulator only; it does not work with the development board version of the debugger. If you attempt to use the RUNF command with a development board, the debugger displays this error message:

```
        --- Execution error
```

---

scolor	*Change Screen Colors*

Syntax   **scolor**   *area name*, *attribute$_1$* [, *attribute$_2$* [, *attribute$_3$* [, *attribute$_4$*]]]

Menu selection   **C**olor→**C**onfig

Description   The SCOLOR command changes the color of specified areas of the debugger display and updates the display immediately. The *area name* parameter identifies the area of the display that is affected. The *attributes* identify how the area is affected. The first two *attribute* parameters usually specify the foreground and background colors for the area. If you do not supply a background color, the debugger uses black as the background.

Valid values for the *attribute* parameters include:

black	blue	green	cyan
red	magenta	yellow	white
bright		blink	

Valid values for the *area name* parameters include:

menu_bar	menu_border	menu_entry	menu_cmd
menu_hilite	menu_hicmd	win_border	win_hiborder
win_resize	field_text	field_hilite	field_edit
field_label	field_error	cmd_prompt	cmd_input
cmd_cursor	cmd_echo	asm_data	asm_cdata
asm_label	asm_clabel	background	blanks
error_msg	file_line	file_eof	file_text
file_brk	file_pc	file_pc_brk	

You don't have to type an entire *attribute* or *area name*; you need to type only enough letters to uniquely identify the attribute. If you supply ambiguous *attribute* names, the debugger interprets the names in this order: black, blue, bright, blink. If you supply ambiguous *area names*, the debugger interprets them in the order that they're listed above (left to right, top to bottom).

---

sconfig   *Load Screen Configuration*

Syntax   **sconfig**   [*filename*]

Menu selection   **C**olor→**L**oad

Description   The SCONFIG command restores the display to a specified configuration. This restores the screen colors, window positions, window sizes, and border styles that were saved with the SSAVE command into *filename*. If you don't supply a *filename*, the debugger looks for the init.clr file. The debugger searches for the specified file in the current directory and then in directories named with the D_DIR environment variable.

setf — *Set Default Data-Display Format*

Syntax	**setf** [*data type*, *display format*]
Menu selection	none

Description   The SETF command changes the display format for a specific data type. If you enter SETF with no parameters, the debugger lists the current display format for each data type.

❑ The *data type* parameter can be any of the following C data types:

char	short	uint	ulong	double
uchar	int	long	ulong	float	ptr

❑ The *display format* parameter can be any of the following characters:

Parameter	Result	Parameter	Result
*	Default for the data type	**x**	Hexadecimal
c	ASCII character (bytes)	**o**	Octal
d	Decimal	**p**	Valid address
e	Exponential floating point	**s**	ASCII string
f	Decimal floating point	**u**	Unsigned decimal

Only a subset of the display formats can be used for each data type. Listed below are the valid combinations of data types and display formats.

Data Type	c	d	o	x	e	f	p	s	u	Data Type	c	d	o	x	e	f	p	s	u
char (c)	√	√	√	√					√	long (d)	√	√	√	√					√
uchar (d)	√	√	√	√					√	ulong (d)	√	√	√	√					√
short (d)	√	√	√	√					√	float (e)			√	√	√	√			
int (d)	√	√	√	√					√	double (e)			√	√	√	√			
uint (d)	√	√	√	√					√	ptr (p)			√	√			√	√	

To return all data types to their default display format, enter:

setf * ⏎

size	*Size Active Window*

Syntax   **size** [*width*, *length*]

Menu selection   none

Description   The SIZE command changes the size of the active window. You can use the SIZE command in one of two ways:

❑ By supplying a specific *width* and *length* or
❑ By omitting the *width* and *length* parameters and using function keys to interactively resize the window.

Valid values for the width and length depend on the screen size and the window position on the screen. These are the minimum and maximum window sizes.

Screen size	Debugger options	Valid widths	Valid lengths
80 characters by 25 lines	none	4 through 80	3 through 24
80 characters by 43 lines (EGA) 80 characters by 50 lines (VGA)	–b	4 through 80	3 through 42 3 through 49
120 characters by 43 lines	–bb	4 through 120	3 through 42
132 characters by 43 lines	–bbb	4 through 132	3 through 42
80 characters by 60 lines	–bbbb	4 through 80	3 through 59
100 characters by 60 lines	–bbbbb	4 through 100	3 through 59

Note: To use larger screen sizes, you must invoke the debugger with the appropriate –b option.

The maximum sizes assume that the window is in the upper left corner (beneath the menu bar). If a window is in the middle of the display, for example, you can't size it to the maximum height and width; you can size it only to the right and bottom screen borders.

If you enter the SIZE command without *width* and *length* parameters, you can use arrow keys to size the window:

⬇⬇   Makes the active window one line longer.
⬆   Makes the active window one line shorter.
⬅   Makes the active window one character narrower.
➡   Makes the active window one character wider.

When you're finished using the arrow keys, you *must* press ⎡ESC⎤ or ⎡➚⎤.

sload	*Load Symbol Table*

Syntax   **sload**   *object filename*

Menu selection   **L**oad→**S**ymbols

Description   The SLOAD command loads the symbol table of the specified object file. SLOAD is useful in a debugging environment in which the debugger cannot, or need not, load the object code (for example, if the code is in ROM). SLOAD clears the existing symbol table before loading the new one but does not modify memory or set the program entry point. Note that SLOAD closes the WATCH and DISP windows.

sound	*Enable Error Beep*

Syntax   **sound   on** | **off**

Menu selection   none

Description   You can cause a beep to sound every time a debugger error message is displayed. This is useful if the COMMAND window is hidden (you wouldn't see the error message). By default, sound is off.

ssave	*Save Screen Configuration*

Syntax   **ssave**   [*filename*]

Menu selection   **C**olor→**S**ave

Description   The SSAVE command saves the current screen configuration to a file. This saves the screen colors, window positions, window sizes, and border styles. The *filename* parameter names the new screen configuration file. You can include path information (including relative pathnames); if you don't supply path information, the debugger places the file in the current directory. If you don't supply a *filename*, then the debugger saves the current configuration into a file named init.clr and places the file in the current directory.

step	*Single-Step*

Syntax	**step** [*expression*]

Menu selection Step=**F8** (in disassembly)

Description The STEP command single-steps through assembly language or C code. If you're in C code, the debugger executes one C statement at a time. In assembly or mixed mode, the debugger executes one assembly language statement at a time.

If you're single-stepping through C code and encounter a function call, the STEP command shows you the single-step execution of the called function (assuming that the function was compiled with the compiler's –g debug option). When function execution completes, single-step execution returns to the caller. If the function wasn't compiled with the debug option, the debugger executes the function but doesn't show single-step execution of the function.

The *expression* parameter specifies the number of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page xv, discusses this in detail).

system	*Enter Operating-System Command*

Syntax	**system** [*operating-system command* [, *flag*]]

Menu selection none

Description The SYSTEM command allows you to enter operating-system commands without explicitly exiting the debugger environment.

If you enter SYSTEM with no parameters, the debugger will open a system shell and display the operating-system prompt. At this point, you can enter any operating-system command. (In MS-DOS, available memory may limit the commands that you can enter.) When you finish, enter the appropriate information to return to the debugger environment:

MS-DOS	*UNIX*	
exit ⏎	**exit** ⏎ **or**	CONTROL D

If you prefer, you can supply the operating-system command as a parameter to the SYSTEM command. If the result of the command is a message or other display, the debugger will blank the top of the debugger display to show the information. In this case, you can use the *flag* parameter to tell the debugger whether or not it should hesitate after displaying the information. *Flag* may be a 0 or a 1.

0   If you supply a value of 0 for *flag*, the debugger immediately returns to the debugger environment after the last item of information is displayed.

1   If you supply a value of 1 for *flag*, the debugger does not return to the debugger environment until you press ⏎. (This is the default.)

take	*Execute Batch File*

Syntax   **take**   *batch filename*   [*, suppress echo flag*]

Menu selection   none

Description   The TAKE command tells the debugger to read and execute commands from a batch file. The *batch filename* parameter identifies the file that contains commands.

By default, the debugger echoes the commands to the output area of the COMMAND window and updates the display as it reads the commands from the batch file. This behavior can be changed by using the *suppress echo flag*:

❏ If you don't use the *suppress echo flag* parameter, or if you use it but supply a nonzero value, then the debugger behaves in the default manner.

❏ If you would like to suppress the echoing and updating, use the value 0 for the *suppress echo flag* parameter.

tba　　　　　　　　*Set a Tentative Breakpoint on a TIGA Module*

Syntax　　　　　　**tba**　*function name*

Menu selection　　none

Description　　　　The TBA command tells the debugger to look for *function name* in the TIGA application and set a breakpoint on that function when the module is loaded by the TIGA application.

tbd　　　　　　　　*Clear a Tentative Breakpoint*

Syntax　　　　　　**tbd**　*breakpoint index*

Menu selection　　none

Description　　　　The TBD command deletes a tentative breakpoint from the list of functions to be watched for. The *breakpoint index* parameter corresponds to the index numbers shown with the BL (breakpoint list) command. If a tentative breakpoint is resolved and active, it cannot be deleted with this command; instead, you must use the BR (breakpoint reset) command.

unalias　　　　　　*Remove Custom Command String*

Syntax　　　　　　**unalias**　*alias name*

Menu selection　　none

Description　　　　The UNALIAS command deletes an alias and its definition.

use	*Use Different Directory*

Syntax   **use**   *directory name*

Menu selection   none

Description   The USE command names an additional directory that the debugger can search when looking for source files. You can specify only one directory at a time.

wa	*Watch Value Add*

Syntax   **wa**   *expression* [*, label*] [*, display format*]

Menu selection   **W**atch→**A**dd

Description   The WA command displays the value of *expression* in the WATCH window. If the WATCH window isn't open, executing WA opens the WATCH window. The *expression* parameter can be any C expression, including an expression that has side effects.

It's most useful to watch an expression whose value changes over time; constant expressions provide no useful function in the watch window. The *label* parameter is optional. When used, it provides a label for the watched entry. If you don't use a *label*, the debugger displays the *expression* in the label field.

When you use the optional *display format* parameter, data will be displayed in one of the following formats:

Parameter	Result	Parameter	Result
*	Default for the data type	**x**	Hexadecimal
c	ASCII character (bytes)	**o**	Octal
d	Decimal	**p**	Valid address
e	Exponential floating point	**s**	ASCII string
f	Decimal floating point	**u**	Unsigned decimal

If you want to use a *display format* parameter without a *label* parameter, just insert an extra comma. For example:

```
wa PC,,d
```

wd   *Watch Value Delete*

Syntax   **wd**   *index number*

Menu selection   **W**atch→**D**elete

Description   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.

whatis   *What Is This Data?*

Syntax   **whatis**   *symbol*

Menu selection   none

Description   The WHATIS command shows the data type of *symbol* in the COMMAND window display area. The *symbol* can be any variable (local, global, or static), a function name, structure tag, typedef name, or enumeration constant.

win   *Select Active Window*

Syntax   **win**   *WINDOW NAME*

Menu selection   none

Description   The WIN command allows you to select the active window by name. Note that the *WINDOW NAME* is in uppercase (matching the name exactly as displayed). You can spell out the entire window name, but you really need to specify only enough letters to identify the window.

            If several of the same types of window are visible on the screen, don't use the WIN command to select one of them. If you supply an ambiguous name (such as C, which could stand for CPU or CALLS), the debugger selects the first window it finds whose name matches the name you supplied. If the debugger doesn't find the window you asked for (because you closed the window or misspelled the name), then the WIN command has no effect.

wr	*WATCH Window Reset*

Syntax   **wr**

Menu selection   **W**atch→**R**eset

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

zoom	*Enlarge Active Window*

Syntax   **zoom**

Menu selection   none

Description   The ZOOM command makes the active window as large as possible. To
                       "unzoom" a window, enter the ZOOM command a second time; this returns the
                       window to its prezoom size and position.

14.3 Summary of Special Keys

The debugger provides function key, cursor key, and command key sequences for performing a variety of actions:

❑ Editing text on the command line
❑ Using the command history
❑ Switching modes
❑ Halting or escaping from an action
❑ Displaying the pulldown menus
❑ Running code
❑ Selecting or closing a window
❑ Moving or sizing a window
❑ Scrolling through a window's contents
❑ Editing data or selecting the active field

Editing text on the command line

To do this	Use these function keys
Enter the current command (note that if you press the return key in the middle of text, the debugger truncates the input text at the point where you press this key)	⏎
Move back over text without erasing characters	CTRL H or BACK SPACE
Move forward through text without erasing characters	CTRL L
Move back over text while erasing characters	DELETE
Move forward through text while erasing characters	SPACE
Insert text into the characters already on the command line	INSERT

Using the command history

To do this	Use these function keys
Repeat the last command that you entered	F2
Move backward, one command at a time, through the command history	TAB
Move forward, one command at a time, through the command history	SHIFT TAB

Switching modes

To do this	Use this function key
Switch debugging modes in this order:	F3

auto ⟶ assembly ⟶ mixed ⟶

Halting or escaping from an action

The escape key acts as an end or undo key in several situations.

To do this	Use this function key
❏ Halt program execution	ESC
❏ Close a pulldown menu	
❏ Undo an edit of the active field in a data-display window (pressing this key leaves the field unchanged)	
❏ Halt the display of a long list of data in the COMMAND window display area	

Displaying pulldown menus

To do this	Use these function keys
Display the Load menu	ALT L
Display the Break menu	ALT B
Display the Watch menu	ALT W
Display the Memory menu	ALT M
Display the Color menu	ALT C
Display the MoDe menu	ALT D
Display an adjacent menu	← or →
Execute any of the choices from a displayed pulldown menu	Press the highlighted letter corresponding to your choice

Running code

To do this	Use these function keys
Run code from the current PC (equivalent to the RUN command without an *expression* parameter)	F5
Single-step code from the current PC (equivalent to the STEP command without an *expression* parameter)	F8
Single-step code from the current PC; step over function calls (equivalent to the NEXT command without an *expression* parameter)	F10

Selecting or closing a window

To do this	Use these function keys
Select the active window (pressing this key makes each window active in turn; stop pressing the key when the desired window becomes active)	F6
Close the CALLS or DISP window (the window must be active before you can close it)	F4
Repeat the last command	F2

Moving or sizing a window

You can use the arrow keys to interactively move a window after entering the MOVE or SIZE command without parameters.

To do this	Use these function keys
❏ Move the window down one line	↓
❏ Make the window one line longer	
❏ Move the window up one line	↑
❏ Make the window one line shorter	
❏ Move the window left one character position	←
❏ Make the window one character narrower	
❏ Move the window right one character position	→
❏ Make the window one character wider	

Scrolling a window's contents

These descriptions and instructions for scrolling apply to the active window. Some of these descriptions refer to specific windows; if no specific window is named, then the description/instructions refer to any window that is active.

To do this	Use these function keys
Scroll up through the window contents, one window length at a time	[PAGE UP]
Scroll down through the window contents, one window length at a time	[PAGE DOWN]
Move the field cursor up one line at a time	[↑]
Move the field cursor down one line at a time	[↓]
❏ *FILE window only:* Scroll left 8 characters at a time ❏ *Other windows:* Move the field cursor left 1 field; at the first field on a line, wrap back to the last fully displayed field on the previous line	[←]
❏ *FILE window only:* Scroll right 8 characters at a time ❏ *Other windows:* Move the field cursor right 1 field; at the last field on a line, wrap around to the first field on the next line	[→]
FILE window only: Adjust the window's contents so that the first line of the text file is at the top of the window	[HOME]
FILE window only: Adjust the window's contents so that the last line of the text file is at the bottom of the window	[END]
DISP windows only: Scroll up through an array of structures	[CTRL] [PAGE UP]
DISP windows only: Scroll down through an array of structures	[CTRL] [PAGE DOWN]

Editing data or selecting the active field

The F9 function key makes the current field (the field that the cursor is pointing to) active. This has various effects, depending on the field.

To do this	Use this function key
❏ *FILE or DISASSEMBLY window:* Set or clear a breakpoint ❏ *CALLS window:* Display the source to a listed function ❏ *Any data-display window:* Edit the contents of the current field ❏ *DISP window:* Open an additional DISP window to display a member that is an array, structure, or pointer	[F9]

Basic Information
About C Expressions

Many of the debugger commands take C expressions as parameters. This allows the debugger to have a relatively small yet powerful instruction set. Because C expressions can have side effects—that is, the evaluation of some types of expressions can affect existing values—you can use the same command to display or to change a value. This reduces the number of commands in the command set.

This chapter contains basic information that you'll need to know in order to use C expressions as debugger command parameters.

	Synopsis Page	Topic
If you're an experienced C programmer, skip this section.	**15.1 C Expressions for Assembly Language Programmers**	**ii**
Because the C expressions you'll use are parameters to debugger commands, some language features may be inappropriate. This section covers specific implementation issues (including necessary limitations and additional features) related to using C expressions as command parameters.	**15.2 Restrictions and Features Associated With Expression Analysis in the Debugger** Restrictions iv Additional features	**iv** iv

15.1 C Expressions for Assembly Language Programmers

It's not necessary for you to be an experienced C programmer in order to use the debugger. However, in order to use the debugger's full capabilities, you might find it helpful to be familiar with the rules governing C expressions. You should obtain a copy of ***The C Programming Language*** (first or second edition) by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey. This book is referred to in the C community and in Texas Instruments documentation as **K&R.**

Note:

A single value or symbol is a legal C expression.

K&R contains a complete description of C expressions; to get you started, here's a summary of the operators that you can use in expression parameters.

❑ **Reference operators**

–>	indirect structure reference	.	direct structure reference
[ ]	array reference	*	indirection (unary)
&	address (unary)		

❑ **Arithmetic operators**

+	addition (binary)	–	subtraction (binary)
*	multiplication	/	division
%	modulo	–	negation (unary)
(*type*)	typecast		

❑ **Relational and logical operators**

>	greater than	>=	greater than or equal to
<	less than	<=	less than or equal to
= =	is equal to	!=	is not equal to
&&	logical AND	||	logical OR
!	logical NOT (unary)		

❑ **Increment and decrement operators**

++ increment − − decrement

These unary operators can precede or follow a symbol. When the operator precedes a symbol, the symbol value is incremented/decremented before it is used in the expression; when the operator follows a symbol, the symbol value is incremented/decremented after it is used in the expression. Because these operators affect the symbol's final value, they have side effects.

❑ **Bitwise operators**

&	bitwise AND	|	bitwise OR
^	bitwise exclusive-OR	<<	left shift
>>	right shift	~	1s complement (unary)

❑ **Assignment operators**

=	assignment	+=	assignment with addition
−=	assignment with subtraction	/=	assignment with division
%=	assignment with modulo	&=	assignment with bitwise AND
^=	assignment with bitwise XOR	|=	assignment with bitwise OR
<<=	assignment with left shift	>>=	assignment with right shift
*=	assignment with multiplication		

These operators support a shorthand version of the familiar binary expressions; for example, X = X + Y can be written in C as X += Y. Because these operators affect a symbol's final value, they have side effects.

15.2 Restrictions and Features Associated With Expression Analysis in the Debugger

The debugger's expression analysis is based on C expression analysis. This includes all mathematical, relational, pointer, and assignment operators. However, there are a few limitations, as well as a few additional features not described in K&R C.

Restrictions

The following restrictions apply to the debugger's expression analysis features.

❏ The sizeof operator is not supported.

❏ The comma operator (,) is not supported (commas are used to separate parameter values for the debugger commands).

❏ Function calls and string constants are currently not supported in expressions.

❏ The debugger supports a limited number of type casts—the following forms are allowed.

(*basic type* **)**
(*basic type* ***** *...*__**)**__
([*structure/union/enum* **]** *structure/union/enum tag* **)**
([*structure/union/enum* **]** *structure/union/enum tag* ***** *...* **)**

Note that you can use up to six *****s in a cast.

Additional features

❏ All floating-point operations are performed in double precision using standard widening. (This is transparent.)

❏ All registers can be referenced by name.

❏ Void expressions are legal (treated like integers).

❏ The specification of variables and functions can be qualified with context information. Local variables (including local statics) can be referenced with the expression form:

*function name***.***local name*

This expression format is useful for examining the automatic variables of a function that is not currently being executed. Unless the variable is static,

iv

however, the function must be somewhere in the current call stack. Note that if you want to see local variables from the currently executing function, you need not use this form; you can simply specify the variable name (just as in your C source).

File-scoped variables (such as statics or functions) can be referenced with the expression form:

filename.*function name*
or *filename*.*variable name*

This expression format is useful for accessing a file-scoped static variable (or function) that may share its name with variables in other files.

Note that in this expression, *filename* **does not include** the file extension; the debugger searches the object symbol table for any source filename that matches the input name, disregarding any extension. Thus, if the variable ABC is in file source.c, you can specify it as source.ABC.

These expression forms can be combined into an expression of the form:

filename.*function name*.*variable name*

❏ Any integral or void expression may be treated as a pointer and used with the indirection operator (`*`). Here are several examples of valid use of a pointer in an expression:

```
*123
*A5
*(A2 + 123)
*(I*J)
```

By default, the values are treated as integers (that is, these expressions point to integer values).

❏ Any expression can be typecast to a pointer to a specific type (overriding the default of pointing to an integer, as described above).

Hint: You can use casting with the WA and DISP commands to display data in a desired format.

For example, the expression:

```
*(float *)10
```

treats 10 as a pointer to a floating-point value at location 10 in memory. In this case, the debugger fetches the contents of memory location 10 and treats the contents as a floating-point value. If you use this expression as a parameter for the DISP command, the debugger displays memory contents as an array of floating-point values within the DISP window, beginning with memory location 10 as array member [0].

Note how the first expression differs from the expression

```
(float)*10
```

In this case, the debugger fetches an integer from address 10 and converts the integer to a floating-point value.

You can also typecast to user-defined types such as structures; for example, the expression:

```
((struct STR *)10)->field
```

In this case, the debugger treats memory location 10 as a pointer to a structure of type STR (assuming that a structure is at address 10) and accesses a field from that structure.

Troubleshooting and Technical Notes

This appendix contains troubleshooting information and additional technical information about how the debugger works with the emulator and with development boards. Most people will not need this information. However, it is provided to help you if:

❑ You are having problems invoking the debugger
❑ The debugger does not seem to be operating properly
❑ You are using an advanced design or nonstandard application (if, for example, you have written your own TIGA communication driver) and you need additional information about board operation or system interfaces

In addition, if you need detailed technical information, you can refer to the user's guide or installation guide that accompanied your board.

A.1 Troubleshooting an Emulator Installation

Here are suggested solutions for problems that you may encounter with your new emulator system.

❑ If you invoke the debugger and see this message:

```
CANNOT INITIALIZE TARGET SYSTEM ! !
– Check I/O configuration
– Check cabling and target power
```

use these questions to determine the problem area:

■ Did you reset the emulator board before invoking the debugger? If not, be sure to execute *emurst* before invoking the debugger again. For more information, refer to *Resetting the emulator* on page viii.

■ Are you identifying the correct I/O space? If you modified the emulator's I/O switches, be sure to use the emurst –p option and the db340emu –p option with the correct *port address* parameter. For more information, refer to *Resetting the emulator* on page viii and *Invoking the Debugger* on page xii.

■ Is the cable connected correctly between the emulator and the target system? (If you check this, remember to be very careful with the target cable connector.)

❑ If the debugger comes up correctly but does not recognize an execution instruction (RUN, STEP, etc.), then the HSTCTLH register's HLT bit might be set. Clear the bit by modifying the contents of memory location 0xC000 0100 or by entering this command:

? HLT=0 ⏎

❑ If the contents of memory seem to be incorrect, check the CONFIG register to ensure that you have properly selected little-endian or big-endian mode. During a reset, the four LSBs of the reset vector determine the endian mode and the memory configuration. The target '34020 will not be completely reset until an execution command (RUN, STEP, etc.) is executed. For more information about this type of problem, refer to the *TMS34020 Emulator Installation Guide.*

A.2 Troubleshooting a Development Board Installation

Here are suggested solutions for problems that you may encounter with your new development system.

❏ **CANNOT FIND EVM MONITOR.** This means that the debugger cannot find the gspmon.out file. Be sure that the PATH statement and the D_DIR environment variable were set up to identify the directory that contains gspmon.out, and that the commands to do this have been executed. If you are not sure, execute the DOS SET command without parameters:

set ⏎

This displays the current PATH and environment variable settings. If the settings are incorrect, re-execute the autoexec or initdb.bat file that sets these commands. Correct settings are shown in Section 2.3 on page iv.

❏ **CANNOT LOAD EVM MONITOR.** This means that there is a problem with loading the gspmon.out file onto the development board. Execute the gspsetup utility (see page A-4); gspsetup may report one of these errors with TIGA communications:

   ■ **TIGA communications driver not installed.** This means that tigacd wasn't executed; be sure to invoke the tigacd utility as described in *Installing the TIGA communication driver* on page vii.

   ■ **gspsetup lists memory errors.** If gspsetup lists memory errors, then the development board is using the wrong TIGA communications driver.

❏ **CANNOT RESET EVM.** This means that the debugger has loaded the gspmon.out file onto the development board, but gspmon is not operating correctly. There are two reasons that this might happen:

   ■ If you are also using a '34020 emulator in your system, then the debugger may not be able to gain control of the '34020. Run gspsetup. If it reports that it is not able to gain control of the '340 processor, check to see if an emulator is connected to the target system. If so, refer to Section 1.4 (page ix). If an emulator is not connected to the target system, your board may be inoperable.

   ■ The TIGA communication driver is not installed properly. Run gspsetup to determine if this is the problem.

Common serial link problems

If you are using the development board version of the debugger for debugging TIGA applications, you may experience some of these problems:

❑ **GSPSETUP hangs for several seconds;** the PC beeps three times, then returns to DOS. This means that the serial link is not operating properly. Uninstall the tigacom and debugcom drivers by entering:

tigacom /u ⏎
debugcom /u ⏎

Now reinstall tigacom and debugcom; use a lower baud rate and make sure that you are using correct communication port settings.

❑ **GSPSETUP or DB340 executes properly, but you encounter data errors.** There are two reasons that this may happen:

■ The TIGA communication driver may not be operating properly on the target system. To check it, first, disable the TIGACD debug mode:

tigacd /d0 ⏎

Now enter:

tigalnk /lx ⏎

If no errors are reported, you can assume that the TIGA communication driver is operating properly.

■ You may be using a baud rate that is too high for the host and target systems. Uninstall tigacom and debugcom (as described above), then reinstall them with a lower baud rate.

❑ **DB340 randomly locks up;** the PC beeps three times, then returns to DOS. There are three reasons that this may happen:

■ An application or driver executing on the target system may have corrupted the settings of the COM port (mouse drivers often use a COM port).

■ You may be using a baud rate that is too high for the host and target systems. Uninstall tigacom and debugcom, then reinstall them with a lower baud rate.

■ You may be operating in a 386 protected mode. Latency times due to mode switching result in a lowering of the maximum baud rate that is available in real mode. Try uninstalling debugcom and tigacom, then reinstalling them at a lower baud rate.

❑ **Operation is not smooth** (for example, scrolling is not smooth). Sometimes, the debugger will hang for a couple of seconds, and executing

`gspsetup –d` does not produce smooth output of test results. The most common cause of this is hardware interrupts conflicting with PC time. The best solution is to remove any software that may be causing these conflicts from your target and host PC systems. Types of software that can cause this problem include LAN drivers, print spoolers, and any other serial devices (although a serial mouse has no noticeable effect).

Running the gspsetup utility

The gspsetup utility is used for verifying a proper serial link, but it can also be used for other purposes. The general format for running gspsetup is:

gspsetup [–*options*]

The gspsetup program has several options; these are listed in Table A–1.

Table A–1. gspsetup Options

Option	Description
–T	Test TIGA's host-to-'340 memory interface (this helps you to be sure that the TIGA communications driver is operating correctly).
–t	Don't perform tests on memory interface (this is the opposite of –T).
–D	Display information about tests and '340 settings.
–d	Don't display information about tests and '340 settings (this is the opposite of –D).
–M*address* **–m***address*	Tests memory at the specified *address.* This option is useful if you've written your own version of a TIGA communication driver. Use –m to check that the three 32-bit words beginning at *address* are valid. If they are not valid, then gspsetup will display a message stating that it is unable to communicate with the development board via TIGA.
–H, –h, or **–?**	Display gspsetup options.

For most applications, you can invoke gspsetup without options:

gspsetup ⏎

When you do this, gspsetup:

❑ Tests TIGA's host-to-'340 memory interface (as if you had used –T),
❑ Turns the information display off (as if you had used –d), and
❑ Checks that the '340 is running freely.

A.3 **What the Debugger Does During Invocation**

In some circumstances, you may find it helpful to know the steps that the debugger goes through during the invocation process. These are the steps, in order, that the debugger performs when you invoke it.

1) Reads options from the command line.

2) Reads any information specified with the D_OPTIONS environment variable.

3) Reads information from the D_DIR and D_SRC environment variables.

4) This step depends on whether you're using the development board or emulator version of the debugger:

❑ *Development board version:* Initializes the target and downloads gspmon.

❑ *Emulator version:* Resets the target system.

5) Looks for the init.clr screen configuration file (the debugger searches for this file in directories named with D_DIR).

6) Initializes the debugger screen and windows but initially displays only the COMMAND window.

7) Looks for the dbinit.cmd or emuinit.cmd batch file (the debugger searches for this file in directories named with D_DIR). If the debugger finds the file, it opens the file and reads and executes the commands it finds inside.

8) Loads any object filenames specified with D_OPTIONS or specified on the command line during invocation.

9) Determines the initial mode (auto, assembly, or mixed) and displays the appropriate windows on the screen.

At this point, the debugger is ready to process any commands that you enter.

A.4 Using the Emulator With Target Systems That Hold $\overline{\text{HCS}}$ Inactive During Power-Up

> **Note:**
>
> The information in this section applies **only** to target systems that hold $\overline{\text{HCS}}$ inactive during power-up. This includes the '34020 SDB.

In systems that hold $\overline{\text{HCS}}$ inactive during power-up, the HSTCTLH register's HLT bit is set whenever the debugger resets the '34020. This causes the '34020 to remain in a halt state until HLT is cleared. As a result, none of the debugger execution commands (RUN, STEP, etc.) will advance execution of your program. You can fix this situation by performing these steps:

1) Invoke the debugger.

2) Modify the reset vector (address 0xFFFF FFE0). If you have a reset routine, modify the reset vector to point to the beginning this routine:

   ? *0xFFFFFFE0 = *address of your reset routine* ⏎

   Be sure that the four LSBs of the new value indicate little endian/big endian usage and RCA bus configuration (as appropriate).

   If your program doesn't have a reset routine, modify the reset vector to point to c_int00:

   ? *0xFFFFFFE0 = c_int00 ⏎

3) Clear the HLT bit:

   ? HLT=0 ⏎

4) Execute a single instruction:

   STEP ⏎

   The '34020 reset will now take effect:

   ❑ The 28 MSBs of the program counter are set to the contents of the 28 MSBs of the reset vector.

   ❑ The 4 LSBs of the CONFIG register are set to the 4 LSBs of the reset vector.

5) Set the CONFIG register to an appropriate value. For example, if you are using the '34020 SDB as a target system, set CONFIG to 0x0C0A:

   ? CONFIG=0x0C0A ⏎

A batch file named emuinit.cmd is provided to help you perform these steps.

A.5   Debugger and Monitor Communications (Development Boards Only)

The debugger is made up of two separate programs:

❑ The first program, called the *host debugger*, is executed by the host PC.

❑ The second program, called the *target monitor*, is downloaded to the development board by the debugger. The host debugger uses the monitor to control the '340 processor on the development board.

These two programs communicate through TIGA, using an area of development board memory for passing commands and data back and forth.

In most cases, you need not be concerned with the communication between the debugger and the target monitor. However, if you are debugging a sophisticated source program (such as one that modifies certain trap vectors or uses the '34020's single-step capabilities), then it may be necessary to have a greater understanding of this communication process.

The target monitor is in a file named gspmon.out. Whenever you invoke the debugger, it requests TIGA for a 4K–byte area of memory on the TIGA board.

If the target monitor is accidentally modified by a write to these memory locations, the debugger attempts to repair the target monitor and displays an error message in the COMMAND window. If the debugger does not operate correctly after this occurs, then it was unable to repair the target monitor. You must reload the target monitor by entering the RESET and RESTART commands.

If the target monitor was corrupted because the '340 stack used this memory, exit the debugger and reinvoke it.

The target monitor gains control of the '340 by having the host temporarily halt the '340 processor, changing the NMI vector to point to the beginning of the monitor code, sending an NMI to the '340, and then unhalting the '340. The '340 then immediately saves its current context (the contents of the PC and ST) and jumps to the NMI routine (which now points to the monitor program). The target monitor can then store the '340 CPU registers and other information in development board memory, where the host can access them.

When the debugger wants to resume execution of the source program, it signals the target monitor to return from the NMI interrupt routine. The saved PC and ST values are restored; this causes the source code to be executed, and the '340 runs freely again.

Because the '340 needs to save its context on the stack, the data on the stack below the top of the stack is invalid. The target monitor uses six 32-bit memory locations:

Stack Area

valid	**high '340 memory**
valid	
valid	
invalid	
invalid	
invalid	**low '340 memory**

SP →

6 32-bit
memory locations
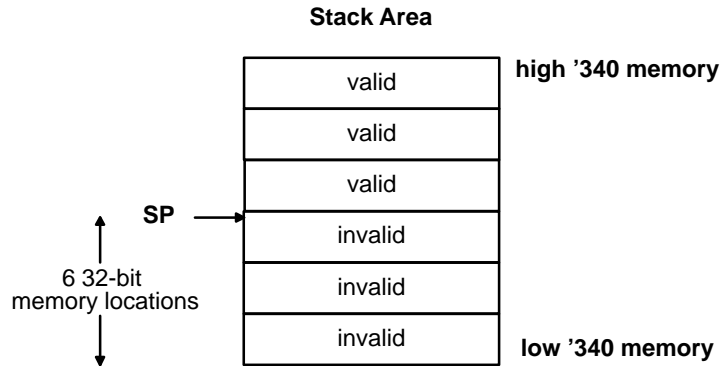
When breakpoints are set in the source program, the debugger temporarily replaces the program instruction at that memory address with a TRAP 29 instruction. (TRAP 29 must be reserved for the debugger's use.) The debugger changes the TRAP 29 vector to point to the target monitor code. When the '340 executes the TRAP 29 instruction, it saves the current context and jumps to the monitor code, which indicates to the host that a breakpoint was encountered. It is important that the program being debugged does not modify the TRAP vector used for breakpoints.

When the debugger is used with a '34020, it uses the single-step status bit (SS) when it is necessary to execute only a single '34020 instruction. The source program should not modify the SS bit or the single-step vector (TRAP 32). The '34010 mimics this single-step ability by examining the next instruction to be executed and placing temporary breakpoints on all the possible PC destinations. The '34020 will also use this technique when it is about to single-step an instruction that could save or modify the status register. However, the debugger cannot anticipate when an interrupt is about to occur. Thus, when a run command is executing, it is possible that an interrupt could occur, and a copy of the ST with a set SS bit could be stored on the stack. This will not cause problems for the debugger; however, it may cause problems for the source program if it saves that ST and later uses it after the debugger is exited.

A.6 Using a TIGA Communication Driver (Development Boards Only)

The development board version of the C source debugger uses TIGA as a software interface between the host PC and the development board. The TIGA communication driver is responsible for providing all hardware-specific communication routines. Therefore, the debugger can be used without restriction for almost any development board—provided that a TIGA communication driver is available.

Use of the communication driver may differ, depending on whether you are using a manufacturer-provided driver or you are developing a driver.

❏ If you already have a TIGA communication driver, load and configure it according to the manufacturer's instructions. Then execute the gspsetup utility (described in Chapter 2). gspsetup tests to see that the host-to-'340 interface is operating correctly. (Note that gspsetup does not test for TIGA compatibility and does not perform an extensive memory test.)

❏ If you do not intend to support TIGA, you can use utilities that come with the debugger to create a communication driver that duplicates a small subset of TIGA's functionality. Follow these steps:

1) Be sure that you have copied the directory \db\hll from the original debugger product diskette into the directory c:\db\hll.

   You may need to modify the source files contained in this directory so that they will better suit your application. Refer to the README file and the source files for more information.

2) In order to create this driver, you will need the Microsoft C compiler (version 5.1 or higher) and Microsoft assembler MASM (version 5.0 or higher), or compatible tools. When these tools are installed, make the c:\db\hll directory the current directory, then enter:

   make ⏎

   This creates an executable file called *hllcd.exe*. The executable form of the driver is a small terminate-and-stay-resident program that you install in PC memory before invoking the debugger. To install hllcd in resident memory, enter:

   hllcd ⏎

3) The driver will install itself in memory and make use of a single interrupt vector in the PC interrupt vector table. The default interrupt vector is 0x7F, but this vector can be changed by adding the following line to your autoexec.bat:

   SET TIGA=–i0x*nn*

   where *nn* represents the hexadecimal value of the desired interrupt vector.

If necessary, you can remove the communication driver from memory by entering:

```
hllcd /u
```

The README file that accompanies the debugger software describes the files and utilities that are used for creating the communication driver. It also describes the routines that are supported.

Debugger Messages

This appendix contains an alphabetical listing of the progress and error messages that the debugger might display in the COMMAND window display area. Each message contains both a description of the situation that causes the message and an action to take if the message indicates a problem or error.

B.1 Associating Sound With Error Messages

You can associate a beeping sound with the display of error messages. To do this, use the SOUND command. The format for this command is:

sound   on | off

By default, no beep is associated with error messages (SOUND OFF). The beep is helpful if the COMMAND window is hidden behind other windows.

B.2 Alphabetical Summary of Debugger Messages

Symbols

']' expected

Description   This is an expression error—it means that the parameter contained an opening **[** symbol but didn't contain a closing **]** symbol.

Action   See Section B.3 (page B-19).

')' expected

Description   This is an expression error—it means that the parameter contained an opening **(** symbol but didn't contain a closing **)** symbol.

Action   See Section B.3 (page B-19).

A

Aborted by user

Description   The debugger halted a long COMMAND display listing (from WHATIS, DIR, ML, or BL) because you pressed the ⎋ESC key.

Action   None required; this is normal debugger behavior.

B

Breakpoint already exists at *address*

Description   During single-step execution, the debugger attempted to set a breakpoint where one already existed. (This isn't necessarily a breakpoint that you set—it may have been an internal breakpoint that was used for single-stepping).

Action   None should be required; you may want to reset the program entry point (RESTART) and re-enter the single-step command.

Breakpoint table full

Description   200 breakpoints are already set and there was an attempt to set another. Note that the maximum limit of 200 breakpoints includes internal breakpoints that the debugger may set for single-stepping. Under normal conditions, this should not be a problem; it is rarely necessary to set this many breakpoints.

Action   Enter a BL command to see where you have breakpoints set in your program. Use the BR command to delete all breakpoints or use the BD command to delete individual unnecessary breakpoints.

C

Cannot allocate host memory

Description   This is a fatal error—it means that the debugger is running out of memory to run in.

Action   You might try invoking the debugger with the –v option so that fewer symbols will be loaded. Or you might want to relink your program and link in fewer modules at a time.

Corrupt call stack

Description   The debugger tried to update the CALLS window and couldn't. This may be because a function was called that didn't return. Or, it could be that the call stack was overwritten in memory.

Action   If your program called a function that didn't return, then this is normal behavior (as long as you intended for the function not to return). Otherwise, you may be overwriting program memory.

Cannot change directory

Description   The directory name specified with the CD command either doesn't exist or is not in the current or auxiliary directories.

Action   Check the directory name that you specified. If this is really the directory that you want, re-enter the CD command and specify the entire pathname for that directory (for example, specify `C:\sdb`, not just `sdb`).

Cannot edit field

Description   Expressions that are displayed in the WATCH window cannot be edited.

Action   If you attempted to edit an expression in the WATCH window, you may have actually wanted to change the value of a symbol or register used in the expression. Use the ? or EVAL command to edit the actual symbol or register. The expression value will automatically be updated.

Cannot find/open initialization file

Description   The debugger can't find the init.cmd file.

Action   Be sure that init.cmd is in the sdb (for development boards) or emu34020 (for the emulator) directory. If it isn't, copy it from the debugger product diskette. If init.cmd is in the correct directory, verify that the D_DIR environment variable is set up to identify the sdb or emu34020 directory. See *Setting Up the Debugger Environment* in the appropriate installation chapter.

Cannot halt the processor

Description   This is a fatal error—for some reason, pressing ⌷ESC⌷ didn't halt program execution.

Action   Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger again.

Cannot map into reserved memory: *address*

Description   Memory at address 0xFFFF A500 through 0xFFFF D000 has been reserved for the monitor program (gspmon).

Action   Do not use the MA command to map into this reserved space.

Cannot open config file

Description   The SCONFIG command can't find the screen-customization file that you specified.

Action   Be sure that the filename was typed correctly. If it wasn't, re-enter the command with the correct name. If it was, re-enter the command and specify full path information with the filename.

Cannot open "*filename*"

Description   The debugger attempted to show *filename* in the FILE window but could not find the file.

Action   Be sure that the file exists as named. If it does, enter the USE command to identify the file's directory.

Cannot open object file: "*filename*"

Description   The file specified with the LOAD, SLOAD, or RELOAD command is not an object file that the debugger can load.

Action   Be sure that you're loading an actual object file. Be sure that the file was linked (you may want to run gspcl again to create an executable object file).

Cannot open new window

Description A maximum of 127 windows can be open at once. The last request to open a window would have made 128, which isn't possible.

Action Close any unnecessary windows. Windows that can be closed include WATCH, CALLS, and DISP. To close the WATCH window, enter WD. To close the CALLS window or a DISP window, make the desired window active and press F4 .

Cannot read processor status

Description This is a fatal error—for some reason, pressing ESC didn't halt program execution.

Action Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger again. If you are using the emulator, check the cable connections, also.

Cannot reset the processor

Description This is a fatal error—for some reason, pressing ESC didn't halt program execution.

Action Exit the debugger. If you are using the development board, invoke the autoexec or initdb.bat file, then invoke the debugger again. If you are using the emulator, there may be a problem with the target system; check the cable connections.

Cannot restart processor

Description If a program doesn't have an entry point, then RESTART won't reset the PC to the program entry point.

Action Don't use RESTART if your program doesn't have an explicit entry point.

Cannot set/verify breakpoint at *address*

Description Either you attempted to set a breakpoint in read-only or protected memory, or there are hardware problems with the target system or development board.

Action Check your memory map. If the address that you wanted to breakpoint wasn't in ROM, see Section B.4 (page B-19).

Cannot step

❑ **If you're using a development board:**

Description The monitor program has been overwritten or damaged by program execution.

Action Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger.

❑ **If you're using an emulator:**

Description There is a problem with the target system.

Action See Section B.4 (page B-19).

Cannot take address of register

Description This is an expression error. C does not allow you to take the address of a register.

Action See Section B.3 (page B-19).

Command "*cmd*" not found

Description The debugger didn't recognize the command that you typed.

Action Re-enter the correct command. Refer to Chapter 14 or the Quick Reference Card for a list of valid debugger commands.

Command timed out, emulator busy

❑ **If you're using a development board:**

Description The monitor program has been overwritten or damaged by program execution.

Action Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger.

❑ **If you're using an emulator:**

Description There is a problem with the target system.

Action See Section B.4 (page B-19).

Conflicting map range

Description   A block of memory specified with the MA command overlaps an existing memory map entry. Blocks cannot overlap.

Action   Use the ML command to list the existing memory map; this will help you find the existing block that the new block would over-lap. If the existing block is not necessary, delete it with the MD command and re-enter the MA command. If the existing block is necessary, re-enter the MA command with parameters that will not overlap the existing block.

E

Emulator I/O address is invalid

Description   The debugger was invoked with the –p option, and an invalid *port address* was used.

Action   For valid *port address* values, refer to Table 3–3 (page xii).

Error in expression

Description   This is an expression error.

Action   See Section B.3 (page B-19).

Execution error

❑ **If you're using a development board:**

Description The monitor program has been overwritten or damaged by program execution.

Action   Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger.

❑ **If you're using an emulator:**

Description There is a problem with the target system.

Action   See Section B.4 (page B-19).

F

File not found

Description   The filename specified for the FILE command was not found in the current directory or any of the directories identified with D_SRC.

Action   Be sure that the filename was typed correctly. If it wasn't, re-enter the FILE command with the correct name. If it was, re-enter the FILE command and specify full path information with the filename.

File not found : "*filename*"

Description   The filename specified for the LOAD, RELOAD, SLOAD, or TAKE command was not found in the current directory or any of the directories identified with D_SRC.

Action   Be sure that the filename was typed correctly. If it wasn't, re-enter the command with the correct name. If it was, re-enter the command and specify full path information with the filename.

File too large (*filename*)

Description   You attempted to load a file that was more than 65,518 bytes long.

Action   Try loading the file without the symbol table (SLOAD), or use gspcl to relink the program with fewer modules.

Float not allowed

Description   This is an expression error—a floating-point value was used invalidly.

Action   See Section B.3 (page B-19).

Function required

Description   The parameter for the FUNC command must be the name of a function in the program that is loaded.

Action   Re-enter the FUNC command with a valid function name.

I

Illegal cast

Description   This is an expression error—the expression parameter uses a cast that doesn't meet the C language rules for casts.

Action   See Section B.3 (page B-19).

Illegal left hand side of assignment

Description   This is an expression error—the left hand side of an assignment expression doesn't meet C language assignment rules.

Action   See Section B.3 (page B-19).

Illegal operand of &

Description   This is an expression error—the expression attempts to take the address of an item that doesn't have an address.

Action   See Section B.3 (page B-19).

Illegal pointer math

Description   This is an expression error—some types of pointer math are not valid in C expressions.

Action   See Section B.3 (page B-19).

Illegal pointer subtraction

Description   This is an expression error—the expression attempts to use pointers in a way that is not valid.

Action   See Section B.3 (page B-19).

Illegal structure reference

Description   This is an expression error—either the item being referenced as a structure probably is not a structure, or you are attempting to reference a nonexistent portion of a structure.

Action   See Section B.3 (page B-19).

Illegal use of structures

Description   This is an expression error—the expression parameter is not using structures according to the C language rules.

Action   See Section B.3 (page B-19).

Illegal use of void expression

Description   This is an expression error—the expression parameter does not meet the C language rules.

Action   See Section B.3 (page B-19).

Integer not allowed

Description   This is an expression error—the command does will not accept an integer as a parameter.

Action   See Section B.3 (page B-19).

Invalid address
—— Memory access outside valid range: *address*

Description   The debugger attempted to access memory at *address*, which is outside the memory map.

Action   Check your memory map to be sure that you access valid memory.

Invalid argument

Description   One of the command parameters does not meet the requirements for the command.

Action   Re-enter the command with valid parameters. Refer to the appropriate command description in Chapter 14.

Invalid attribute name

Description   The COLOR and SCOLOR commands accept a specific set of area names for their first parameter. The parameter entered did not match one of the valid attributes.

Action   Re-enter the COLOR or SCOLOR command with a valid area name parameter. Valid area names are listed in Table 4–2 (page iii).

Invalid color name

Description   The COLOR and SCOLOR commands accept a specific set of color attributes as parameters. The parameter entered did not match one of the valid attributes.

Action   Re-enter the COLOR or SCOLOR command with a valid color parameter. Valid color attributes are listed in Table 4–1 (page ii).

Invalid memory attribute

Description   The third parameter of the MA command specifies the type, or attribute, of the block of memory that MA adds to the memory map. The parameter entered did not match one of the valid attributes.

Action   Re-enter the MA command. Use one of the following valid parameters to identify the memory type:

R, ROM, READONLY	(read-only memory)
W, WOM, WRITEONLY	(write-only memory)
RW, RAM	(read/write memory)
PROTECT	(no-access memory)

Invalid object file

Description   The file specified with the LOAD, SLOAD, or RELOAD command either is not an object file that the debugger can load, or it has been corrupted.

Action   Be sure that you're loading an actual object file. Be sure that the file was linked (you may want to run gspcl again to create an executable object file). If the file you attempted to load was a valid executable object file, then it was probably corrupted; recompile, assemble, and link with gspcl.

Invalid watch delete

Description   The debugger can't delete the parameter supplied with the WD command. Usually, this is because the watch index doesn't exist or because a symbol name, instead of a watch index, was typed.

Action   Re-enter the WD command. Be sure to specify the watch index that matches the item you'd like to delete (this is the number in the left column of the WATCH window). Remember, you can't delete items symbolically—you must delete them by number.

Invalid window position

Description   The debugger can't move the active window to the XY position specified with the MOVE command. Either the XY parameters are not within the screen limits, or the active window may be too large to move to the desired position.

Action   You can use the mouse to move the window.

If you don't have a mouse, enter the MOVE command without parameters; then use the arrow keys to move the window. When you finish, you *must* press ⎡ESC⎦ or ⏎.

If you prefer to use the MOVE command with parameters, refer to Table 6–2 (page xxvii) for a list of the XY limits. The minimum XY position is 0,1; the maximum position depends on which screen size you're using.

Invalid window size

Description   The width and length specified with the SIZE or MOVE command may be too large or too small. If valid width and length were specified, then the active window is already at the far right or bottom of the screen and so cannot be made larger.

Action   You can use the mouse to size the window.

If you don't have a mouse, enter the SIZE command without parameters; then use the arrow keys to move the window. When you finish, you *must* press ⎡ESC⎦ or ⏎.

If you prefer to use the SIZE command with parameters, refer to Table 6–1 (page xxiv) for a list of valid sizes. The minimum size is 4 by 3; the maximum size depends on which screen size you're using.

L

Load aborted

Description   This message always follows another message.

Action   Refer to the message that preceded *Load aborted*.

Lost power (or cable disconnected)

Description   Either the target cable is disconnected, or the target system is faulty.

Action   Check the target cable connections. If the target seems to be connected correctly, see Section B.4 (page B-19).

Lost processor clock

Description   Either the target cable is disconnected or the target system is faulty.

Action   Check the target cable connections. If the target seems to be connected correctly, see Section B.4 (page B-19).

Lval required

Description   This is an expression error—an assignment expression was entered that requires a legal lefthand side.

Action   See Section B.3 (page B-19).

N

Name "*name*" not found

Description   The command cannot find the object named *name*.

Action   If *name* is a symbol, be sure that it was typed correctly. If it wasn't, re-enter the command with the correct name. If it was, then be sure that the associated object file is loaded.

                 If *name* was some other type of parameter, refer to the command's description for a list of valid parameters.

M

Memory access error at *address*

❏ **If you're using a development board:**

Description There is a problem with the TIGA communication driver.

Action If you're using tigacd, exit the debugger. Reinstall tigacd and invoke the debugger. If you're using your own version of a TIGA communication driver, be sure to follow the instructions in Appendix A.

❏ **If you're using the emulator:**

Description Either the processor is receiving a bus fault, or there are problems with target system memory.

Action See Section B.4 (page B-19).

Memory map table full

Description Too many blocks have been added to the memory map. This will rarely happen unless someone is adding blocks word by word (which is inadvisable).

Action Stop adding blocks to the memory map. Consolidate any adjacent blocks that have the same memory attributes.

P

Pointer not allowed

Description This is an expression error.

Action See Section B.3 (page B-19).

Processor is already running

Description One of the RUN commands was entered while the debugger was running free from the target system.

Action Enter the HALT command to stop the free run, then re-enter the desired RUN command.

R

Register access error

❏ **If you're using a development board:**

Description There is a problem with the TIGA communication driver.

Action If you're using tigacd, exit the debugger. Reinstall tigacd and invoke the debugger. If you're using your own version of a TIGA communication driver, be sure to follow the instructions in Appendix A.

❏ **If you're using the emulator:**

Description Either the processor is receiving a bus fault, or there are problems with target system memory.

Action See Section B.4 (page B-19).

S

Specified map not found

Description The MD command was entered with an address or block that is not in the memory map.

Action Use the ML command to verify the current memory map. When you are using MD, you can specify only the first address of a defined block.

Structure member not found

Description This is an expression error—an expression references a non-existent structure member.

Action See Section B.3 (page B-19).

Structure member name required

Description This is an expression error—a symbol name followed by a period but no member name.

Action See Section B.3 (page B-19).

Structure not allowed

Description   This is an expression error—the expression is attempting an operation that cannot be performed on a structure.

Action   See Section B.3 (page B-19).

T

Take file stack too deep

Description   Batch files can be nested up to 10 levels deep. Batch files can call other batch files, which can call other batch files, and so on. Apparently, the batch file that you are TAKEing calls batch files that are nested more than 10 levels.

Action   Edit the batch file that caused the error. Instead of calling another batch file from within the offending file, you may want to copy the contents of the second file into the first. This will remove a level of nesting.

Too many breakpoints

Description   200 breakpoints are already set, and there was an attempt to set another. Note that the maximum limit of 200 breakpoints includes internal breakpoints that the debugger may set for single-stepping. Under normal conditions, this should not be a problem; it is rarely necessary to set this many breakpoints.

Action   Enter a BL command to see where you have breakpoints set in your program. Use the BR command to delete all breakpoints, or use the BD command to delete individual unnecessary breakpoints.

Too many memory maps

Description   Too many blocks have been added to the memory map. This will rarely happen unless someone is adding blocks word by word (which is inadvisable).

Action   Stop adding blocks to the memory map. Consolidate any adjacent blocks that have the same memory attributes.

Too many paths

Description   More than 20 paths have been specified cumulatively with the USE command, D_SRC environment variable, and –i debugger option.

Action   If you are entering the USE command before entering another command that has a *filename* parameter, don't enter the USE command. Instead, enter the second command and specify full path information for the *filename*.

W

Window not found

Description   The parameter supplied for the WIN command is not a valid window name.

Action   Re-enter the WIN command. Remember that window names must be typed in uppercase letters. Here are the valid window names; the bold letters show the smallest acceptable abbreviations:

CALLS	**CP**U	**DISP**	**M**EMORY
COMMAND	**DISA**SSEM-BLY	**F**ILE	**W**ATCH

U

User halt

Description   The debugger halted program execution because you pressed the ESC key.

Action   None required; this is normal debugger behavior.

B.3 Additional Instructions for Expression Errors

Whenever you receive an expression error, you should re-enter the command and edit the expression so that it follows the C language expression rules. If necessary, refer to a C language manual such as ***The C Programming Language*** by Brian W. Kernighan and Dennis M. Ritchie.

B.4 Additional Instructions for Hardware Errors

If you continue to receive the messages that send you to this section, this indicates persistent hardware problems.

❑ **Development Boards**

■ The development board memory may not be properly configured—be sure that you are using an appropriate value in the CONFIG register (refer to the *TMS34020 User's Guide*).

■ Check to be sure that the HSTCTLH register's RESET bit is not set while you are using the debugger.

■ Check to see that the stack pointer (register A15) is pointing to valid memory.

■ Check to see that you are refreshing VRAMs correctly (refer to the *TMS34020 User's Guide*).

❑ **Emulator**

■ If a bus fault occurs, the emulator may not be able to access memory.

■ The '34020 must be reset before you can use the emulator. Most target systems reset the '34020 at power-up; your target system may not be doing this.

■ Host accesses or retries that persist for longer than one second will prevent the emulator from accessing memory. Check your host interface code.

■ The development board memory may not be properly configured—be sure that you are using an appropriate value in the CONFIG register (refer to the *TMS34020 User's Guide*).

■ Check to see that you are refreshing VRAMs correctly (refer to the *TMS34020 User's Guide*).

Glossary

A

active window: The window that is currently selected for moving, sizing, editing, closing, or some other function.

aggregate type: A C data type, such as a structure or array, where a variable is composed of multiple variables, called members.

aliasing: A method of customizing debugger commands; aliasing provides a shorthand method for entering often-used command strings.

ANSI C: A version of the C programming language that conforms to the C standards defined by the *American National Standards Institute.*

assembly mode: A debugging mode that shows assembly language code in the DISASSEMBLY and doesn't show the FILE window, no matter what type of code is currently running.

autoexec.bat: A batch file that contains DOS commands for initializing your PC.

auto mode: A context-sensitive debugging mode that automatically switches between shown assembly language code in the DISASSEMBLY window or C code in the FILE window, depending on what type of code is currently running.

B

batch file: Either of two different types of files. One type of batch file contains DOS commands for the PC to execute. A second type of batch file contains debugger commands for the debugger to execute. The PC doesn't execute debugger batch files, and the debugger doesn't execute PC batch files.

benchmarking: A type of program execution that allows you to track the number of CPU cycles consumed by a specific section of code.

breakpoint: A point within your program where execution will halt because of a previous request from you.

C

CALLS window: A window that lists the functions called by your program.

casting: A feature of C expressions that allows you to use one type of data as if it were a different type of data.

children: Windows opened for aggregate types that are themselves members of an aggregate type displayed in the DISP window.

click: Press and release a mouse button without moving the whole mouse.

CLK: A pseudoregister that shows the number of CPU cycles consumed during benchmarking. The value in CLK is valid only after entering a RUNB command but before entering another RUN command.

code-display windows: Windows that show code, text files, or code-specific information. This category includes the DISASSEMBLY, FILES, and CALLS windows.

COFF: *Common Object File Format*. An implementation of the object file format of the same name developed by AT&T. The '340 family compiler, assembler, and linker use and generate COFF files.

command line: The portion of the COMMAND window where you can enter commands.

command-line cursor: Block-shaped cursor that identifies the current character position on the command line.

COMMAND window: A window that provides a display area where you enter commands and where the debugger echoes command entry , shows command output, and lists progress or error messages.

CPU window: A window that displays the contents of '340 on-chip registers, including the program counter, status register, A-file registers, and B-file registers.

current-field cursor: An icon that identifies the current field in the active window.

cursor: An icon on the screen (such as a rectangle or a horizontal line) that is used as a pointing device. The cursor is usually under mouse or keyboard control.

D

data-display windows: Windows for observing and modifying various types of data. This category includes the MEMORY, CPU, DISP, and WATCH windows.

D_DIR: An environment variable that identifies the directory containing the commands and files necessary for running the debugger.

debugger: A window-oriented software interface that helps you to debug '340 programs running on a '340 development board or '34020 emulator.

development board: A PC board that uses a '34010 or '34020 processor and a TIGA communication driver.

disassembly: A reverse assembly of the contents of memory to form assembly language code.

DISASSEMBLY window: A window that displays the disassembly of memory contents.

DISP window: A window that displays the members of an aggregate data type.

display area: The portion of the COMMAND window where the debugger echoes command entry , shows command output, and lists progress or error messages.

D_OPTIONS: An environment variable that you can use for identifying often-used debugger options.

drag: To move the mouse while pressing one of the mouse buttons.

D_SRC: An environment variable that identifies directories containing program source files.

E

EGA: *Enhanced Graphics Adaptor*. An industry standard for video cards.

EISA: *Extended Industry Standard Architecture*. A standard for PC buses.

emulator: A debugging tool that is external to the target system and provides direct control over the '340 processor that is on the target system.

emurst: A utility that resets the emulator.

environment variable: A special system symbol that the debugger uses for finding directories or obtaining debugger options.

F

FILE window: A window that displays the contents of the current C code. The FILE window is primarily intended for displaying C code but can be used to display any text file.

G

gspcl: A shell utility that invokes the '340 family compiler, assembler, and linker to create an executable object file version of your program.

gspmon: The development board monitor program. It contains '340 routines that the debugger uses for controlling the '340 processor.

gspsetup: A utility that tests TIGA communications for the development board version of the debugger.

H

hybrid applications: '340 applications that are split into two parts. One part runs entirely on the '340 processor, a second part runs on the host PC. The two parts communicate during program execution.

I

initdb.bat: As part of normal debugger use, you must enter DOS commands to set up the debugger environment. The most convenient method for doing this is to edit your PC's autoexec.bat file or to create a separate initdb.bat file that is used only for this purpose.

I/O switches: Hardware switches on the '34020 emulator board that identify the PC I/O memory space used for emulator debugger communications.

ISA: *Industry Standard Architecture.* A subset of the EISA standard.

M

memory map: A special set of commands that tells the debugger which areas of memory can and can't be accessed.

MEMORY window: A window that displays the contents of memory.

menu bar: A row of pulldown menu selections, found at the top of the debugger display.

mixed mode: A debugging mode that simultaneously shows both assembly language code in the DISASSEMBLY window and C code in the FILE window.

mouse cursor: Block-shaped cursor that tracks mouse movements over the entire display.

P

PC: Either of two meanings, depending on the context and where it's used in this book: 1) In installation instructions or information relating to hardware and boards, *PC* means *Personal Computer* (as in IBM PC). 2) In general debugger and program-related information, *PC* means *Program Counter*, which is the register that identifies the current statement in your program.

point: To move the mouse cursor until it overlays the desired object on the screen.

port address: The PC I/O memory space that the debugger uses for communicating with the emulator. The port address is selected via switches on the emulator board and communicated to the debugger with the –p debugger option.

pulldown menu: A command menu that is accessed by name or with the mouse from the menu bar at the top of the debugger display.

S

scalar type: A C type where the variable is a single variable itself, not composed of other variables.

scroll: To move the contents of a window up, down, left, or right to view contents that weren't shown.

SDB: *Software Development Board*. A standalone, high-performance, graphics-development board, compatible with the IBM PC-AT ISA bus. Provides an environment for debugging '34020 application software.

side effects: A feature of C expressions in which using an assignment operator in an expression affects the value of one of the components used in the expression.

single-step: A form of program execution that allows you to see the effects of each statement. The program is executed statement by statement; the debugger pauses after each statement to update the data-display windows.

standalone applications: '340 applications that execute entirely on the '340 processor without interacting with the host PC.

symbol table: A file that contains the names of all variables and functions in your '340 program.

T

target system: A '340 board that works with the emulator; the emulator doesn't contain a '340 and so must use a '340 target board. Usually, the target system is a board that you have designed and that you debug using the emulator and debugger.

TDB: *TIGA Development Board*. A standard video display adapter for ISA- and EISA-based PCs; provides an environment for debugging '34010 application software.

TIGA: *Texas Instruments Graphical Architecture*. A software interface standard that allows a host process to communicate with the '340 graphics processor residing on your development board. The current implementation of TIGA is for the PC market and serves as an interface between the '340 and an 80x86 processor running under MS-DOS.

TIGA applications: A variation of a hybrid application in which the part of the program that would be controlled by the '340 is controlled by TIGA.

tigacd: The TIGA communication driver, which is a terminate-and-stay resident program that runs on a PC. It is specific to a particular board and is supplied by the board manufacturer. The communication driver contains functions that are invoked by the application's calls to the application interface. This enables communication via the PC bus to the target '340 board.

TMS34010: The first–generation graphics system processor from Texas Instruments.

TMS34020: The second–generation graphics system processor from Texas Instruments.

TMS340: A collective term used to mean *'34010 or '34020* or *'34010 and '34020.*

TRAP: A '340 processor software interrupt.

V

VGA: *Video Graphics Array*. An industry standard for video cards.

W

WATCH window: A window that displays the values of selected expressions, symbols, addresses, and registers.

window: A defined rectangular area of virtual space on the display.

Index

D

E

G

H

I

N

S

T

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its products 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.

Customers are responsible for their applications using TI components.

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

TI assumes no liability for applications assistance or customer product design. TI does not 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 products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Resale of TI's products or services with _statements different from or beyond the parameters_ stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products. www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Copyright © 2001, Texas Instruments Incorporated