TMS370 Family
C Source Debugger
User's Guide

PRINTED WITH
SOY INK™

TEXAS
INSTRUMENTS

Printed on Recycled Paper

Read This First

What Is This Book About?

This book tells you how to use the TMS370 C source debugger with these debugging tools:

❑ TMS370 XDS/22 emulation system
❑ TMS370 application board

Each tool has its own version of the debugger. These versions operate almost identically; however, the executable files that invoke them are very different. A separate installation book included in your package provides installation information for each tool and operating system. Be sure to install the correct version of the debugger.

How to Use This Manual

The goal of this book is to help you learn how to use the '370 C source debugger. 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 you receive it.

■ Chapter 1 lists the key features of the debugger, describes additional '370 software tools, tells you how to prepare a '370 program for debugging, and provides instructions and options for invoking the debugger.

■ Chapter 2 is a basic tutorial that introduces you to many of the debugger features.

■ Chapter 3 is a more advanced tutorial for use only with the XDS/22 emulation system. It introduces you to the BTT (breakpoint/trace/ timing) and its features.

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

■ The chapters in Part II detail the individual topics that are introduced in the tutorials. For example, Chapter 4 describes all of the debugger's windows and tells you how to move them and size them; Chapter 5 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:
■ Use the appropriate installation chapter in your accompanying installation guide.
■ Complete the tutorials in Chapter 2 and Chapter 3. Note that the BTT tutorial is for use only with the XDS/22.
■ Browse through 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:
■ Use the appropriate installation chapter in your accompanying installation guide.
■ Complete the tutorial in Chapter 2 and Chapter 3. Note that the BTT tutorial is for use only with the XDS/22.
■ Read all of the chapters in Part II.

Notational Conventions

This document uses the following conventions.

❑ The TMS370 processor is referred to as the **'370**.

❑ 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 function keys. 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**

↖   *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 unless you are in an MS Windows environment; 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 you pressed.

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

          Drag. 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.

❑ 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:

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];`
	`}`

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 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:

mem   *expression* [, *display format*]

mem is the command. This command has two parameters, indicated by *expression* and *display format*. The first parameter must be an actual C expression; the second parameter, which identifies a specific display format, 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.

Information About Cautions

This is an example of a caution statement.

A caution statement describes a situation that could potentially damage your software or equipment.

Please read each caution statement carefully.

Related Documentation From Texas Instruments

The following books describe the TMS370 and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Customer Support Center (CRC) at (214) 995–6611. When ordering, please identify the book by its title and literature number.

TMS370 Family XDS/22 Emulation System Installation Guide tells you how to install the C source debugger interface along with the XDS/22 emulation system (using the DOS operating system). It also covers specifications for your emulator, btt, and memory boards.

TMS370 Family Application Board Installation Guide tells you how to install the C source debugger interface along with the application board (using the DOS operating system).

TMS370 8-Bit Microcontroller Family Assembly Language Tools User's Guide (literature number SPNU010) describes the assembly language tools (assembler, linker, and other tools used to develop assembly code), assembler directives, macros, common object file format, and symbolic debugging directives for the '370 family of devices.

TMS370 8-Bit Microcontroller Family Optimizing C Compiler User's Guide (literature number SPNU022) describes the '370 8-bit C compiler. This C compiler accepts ANSI standard C source code and produces assembly language source code for the '370 8-bit family of devices.

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 microcontroller products	Call the CRC[†] : **(214) 995–6611**
	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[†]: **(214) 995–6611**
Ask questions about product operation or report suspected problems	Call the microcontroller hotline: **(713) 274–2370**
Report mistakes in this document or any other TI documentation	Fill out and return the reader response card at: the end of this book, or send your comments to Technical Publications Manager, MS 702 Texas Instruments Incorporated P.O. Box 1443 Houston, Texas 77251–1443

[†] Texas Instruments Customer Response Center

Trademarks

PC-DOS is a trademark of International Business Machines.

MS-DOS and Windows are trademarks of Microsoft Corporation.

Contents

Part I: Hands-On Information

This chapter provides a step-by-step, hands-on demonstration of basic BTT (breakpoint, trace, and timing) features.

Part II: Debugger Description

Describes the default displays, tells you how to switch between assembly language and C debugging, describes the various types of windows on the display, and tells you how to move and size the windows.

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

6   Defining a Memory Map **6-1**

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.

7   Loading, Displaying, and Running Code **7-1**

Tells you how to use the three debugger modes to view the type of source files that you'd like to see, how to load source files and object files, how to run your programs, and how to halt program execution.

8   Managing Data 8-1

Describes the data-display windows and tells you how to edit data (memory contents, register contents, and individual variables).

9   Using Software Breakpoints 9-1

Describes the use of software breakpoints to halt code execution.

10 Customizing the Debugger Display 10-1

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

Tells you how to use the features of the BTT board to accomplish tasks such as setting hardware breakpoints and collecting trace samples.

Part III: Reference Material

14 Basic Information About C Expressions 14-1

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.

A What the Debugger Does During Invocation A-1

Describes the process the debugger follows during invocation.

B Setting Up the Clock B-1

You may want to use a different clock source at some point during the debugging process. This appendix describes the process of selecting a new clock source.

C Debugger Messages C-1

Describes installation, progress, and error messages that the debugger may display.

D Glossary D-1

Defines acronyms and key terms used in this book.

Figures

Tables

Overview of a Code
Development and Debugging System

The '370 C source debugger is an advanced software interface that helps you to develop, test, and refine '370 C programs (compiled with the '370 optimizing ANSI C compiler) and assembly language programs. The debugger is the interface to the '370 in-circuit XDS/22 emulator and the application board.

The chapter provides an overview of the debugger and the debugging process and describes how the debugging process fits in with the overall code development process.

1.1 Description of the '370 C Source Debugger

The '370 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 '370 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 1–1 identifies several features of the debugger display.

Figure 1–1. The Basic Debugger Display
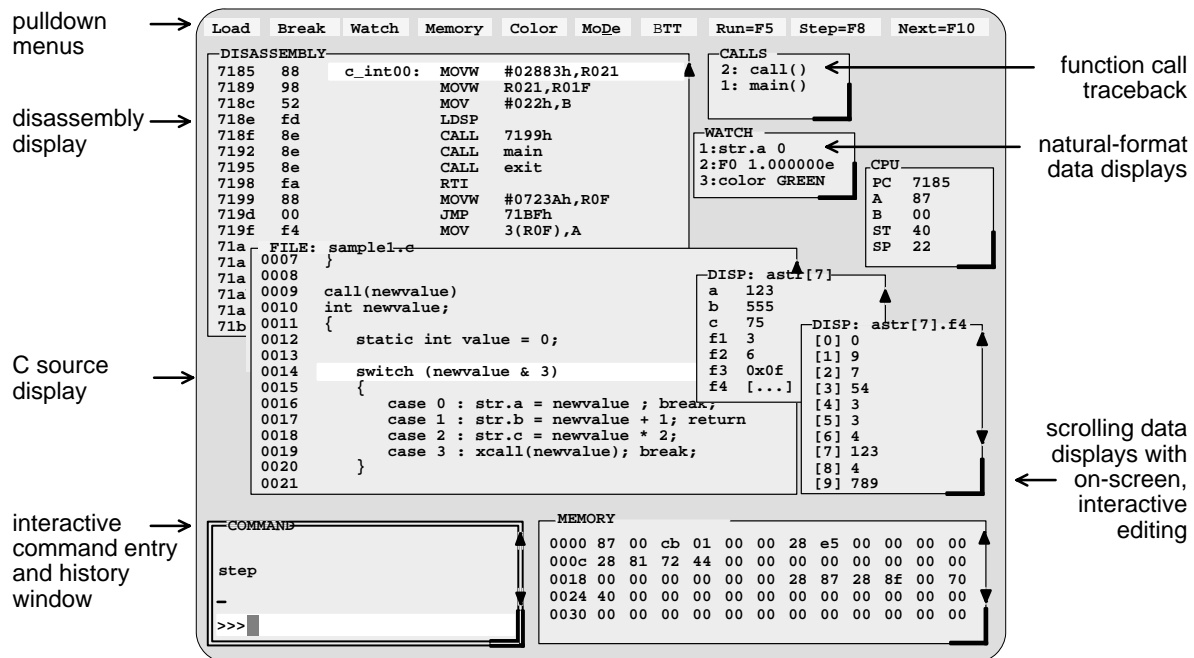
Key features of the debugger

❑ **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   0x0fa│ c   87    ┌─DISP: *str.f3->f3 ─┐
│ f4   [...]│ f1  45    │ a   75            │▲
└───────────│ f2  27    │ b   3212          │
            │ f3  0x0f  │ c   782           │
            │ f4  [...] │ f1  7             │▼
            └───────────│ f2  9             │
                        │ f3  0x0f          │
                        │ f4  [...]         │
                        └───────────────────┘
```
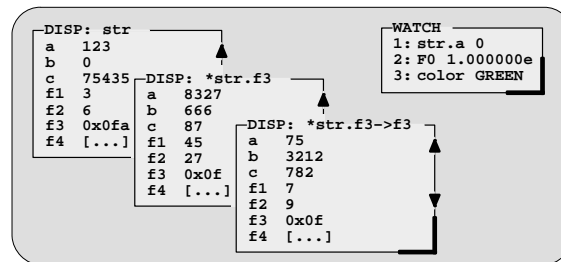
❑ **On-screen editing.** Change any data value displayed in any window—just point the mouse, click, and type.

❑ **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 '370 C source debugger supports a small but powerful command set that makes full use of C expressions. One debugger command performs actions that would require 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.

❏ **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.

Breakpoint, trace, and timing features

Included with the XDS/22 emulation system is a separate board called the BTT board, which provides breakpoint, trace, and timing features. The BTT monitors the '370 CPU; when a preselected pattern of bus activity is detected, the BTT performs an action such as executing a hardware breakpoint or storing information in the trace buffer.

The BTT supports a rich set of features:

❑ **Full range of actions.** The BTT allows you to set hardware breakpoints, to count event occurrences, to collect trace samples, to jump to a BTT state, or to start/stop timers. These actions occur when they are *qualified*—that is, when bus activity matches conditions that you have defined.

❑ **Four separate states.** The BTT supports four separate states, called state 0–state 3. Each state can be associated with up to four actions. You can define actions for as many states as you need. By default, the BTT will cycle through the states, beginning with state 0 and ending with the last state that you defined actions for. You can control this sequencing by jumping to another state or by using counters to loop through a sequence of states.

XDS/22 emulator only

❑ **Flexible qualification for actions.** You can qualify actions according to address or data values (either singly or in relation to ranges) and to the memory-cycle type. You can combine these conditions; for example, you could qualify an action whenever a certain data value is accessed during an instruction acquisition cycle.

❑ **Informative trace reporting.** The BTT can store up to 2047 trace samples in the trace buffer. You can display the trace samples and associated information by opening the INSPECT window.

❑ **External signal access.** The BTT has eight external probes that can be connected to eight signals. You can qualify actions by looking for a particular pattern of activity on these probes. Additionally, the trace buffer reports the values that were on the signals when each trace sample was collected.

❑ **Complete timing analysis.** The BTT supports two timers that you can start and stop on a variety of conditions. The BTT reports the total time for each of these timers; it also reports the average for one of the timers. Additionally, the BTT collects timing information that relates specifically to the samples in the trace buffer.

❑ **External filing.** Once you have defined a complex BTT setup, you may want to reuse the setup. The BTT allows you to save the setup to a file and then load it again for a later session. You can also save the contents of the trace buffer to a file for later use or to compare to another trace collection.

1.2 Description of the Profiling Environment

In addition to the basic debugging environment, a second environment—the *profiling environment*—is available. The profiling environment provides a method for collecting execution statistics about specific areas in your code. This gives you immediate feedback on your application's performance.

Figure 1–2 identifies several features of the debugger display within the profiling environment.

Figure 1–2. The Profiling-Environment Display

profiling areas are clearly marked

pulldown menu provides access to often-used basic debugger commands *plus* special profiling commands

```
 Load    mAp    Mark    Enable    Disable    Unmark    View    Stop-points    Profile
 DISASSEMBLY
 7000  8e          main:      CALL    i0i0it
 7003  42          Re>        MOV     R018,R017
 7006  42                     MOV     R019,R018
 7009  88          <<         MOVW    #00019h,R01B
 700d  8e                     CALL    random
 7010  00                     JMP     main+44(702Ch)
 7012  8e                     CALL    write_number
 7015  00                     JMP     main+3(7003h)
 7017  8e                     CALL    write_number
```

PROFILE window displays execution statistics

```
 PROFILE
      Area Name        Count  Inclusive  Incl-Max  Exclusive  Excl-Max
 CF  main()                1      17450         0       2193         0
 AR  7003-7009            19        589        31        589        31
 CF  write_number()       19       1045        55       1045        55
 CF  random()             19      14022       738        703        37
```

```
 FILE: sample.c
 0034       }
 0035   Fe>  write_number()
 0036        <
 0037   Fe>  results[previous[0]]=rnum;
 0038        >
 0039
 0040   Fe>  random(r)
 0041        int *r;
 0042   Fe>  <
 0043   Fe>  *r=rand()&0xFF;
```

profiling areas are clearly marked

```
 COMMAND
  65 symbols loaded
 Done
 file sample.c
 >>>
```

Key features of the profiling environment

The profiling environment builds on the same easy-to-use interface available in the basic debugging environment and provides these additional features:

❏ **More efficient code.** Within the profiling environment, you can quickly identify busy sections in your programs. This helps you to direct valuable development time at streamlining the sections of code that most dramatically affect program performance.

❏ **Statistics on multiple areas.** You can collect statistics about individual statements in disassembly or C, about ranges in disassembly or C, and about C functions. When you are collecting statistics on many areas, you can choose to view the statistics for all the areas or a subset of the areas.

❏ **Comprehensive display of statistics.** The profiler provides all the information you need for identifying bottlenecks in your code:

■ The number of times each area was entered during the profiling session.

■ The total execution time of an area, including or excluding the execution time of any subroutines called from within the area.

■ The maximum time for one iteration of an area, including or excluding the execution time of any subroutines called from within the area.

Statistics may be updated continuously during the profiling session, or at selected intervals.

❏ **Configurable display of statistics.** Display the entire set of data, or display one type of data at a time. Display all the areas you're profiling, or display a selected subset of the areas.

❏ **Visual representation of statistics.** When you choose to display one type of data at a time, the statistics will be accompanied by histograms for each area, showing the relationship of each area's statistics to those of the other profiled areas.

❏ **Disabled areas.** In addition to identifying areas that you can collect statistics on, you can also identify areas that you don't want to affect the statistics. This removes the timing impact from code such as a standard library function or a fully optimized portion of code.

❏ **Special profiling commands.** The profiling environment supports a rich set of commands to help you select areas and display information. Some of the basic debugger commands—such as the memory map commands—may be necessary during profiling and are available within the profiling environment. Other commands—such as breakpoint commands and run commands—are not necessary and are therefore not available within the profiling environment.

1.3 Developing Code for the '370

The '370 is well supported by a complete set of hardware and software development tools, including a C compiler, assembler, and linker. Figure 1–3 illustrates the '370 code development flow. The figure highlights the most common paths of software development; the other portions are optional.

Figure 1–3. '370 Software Development Flow
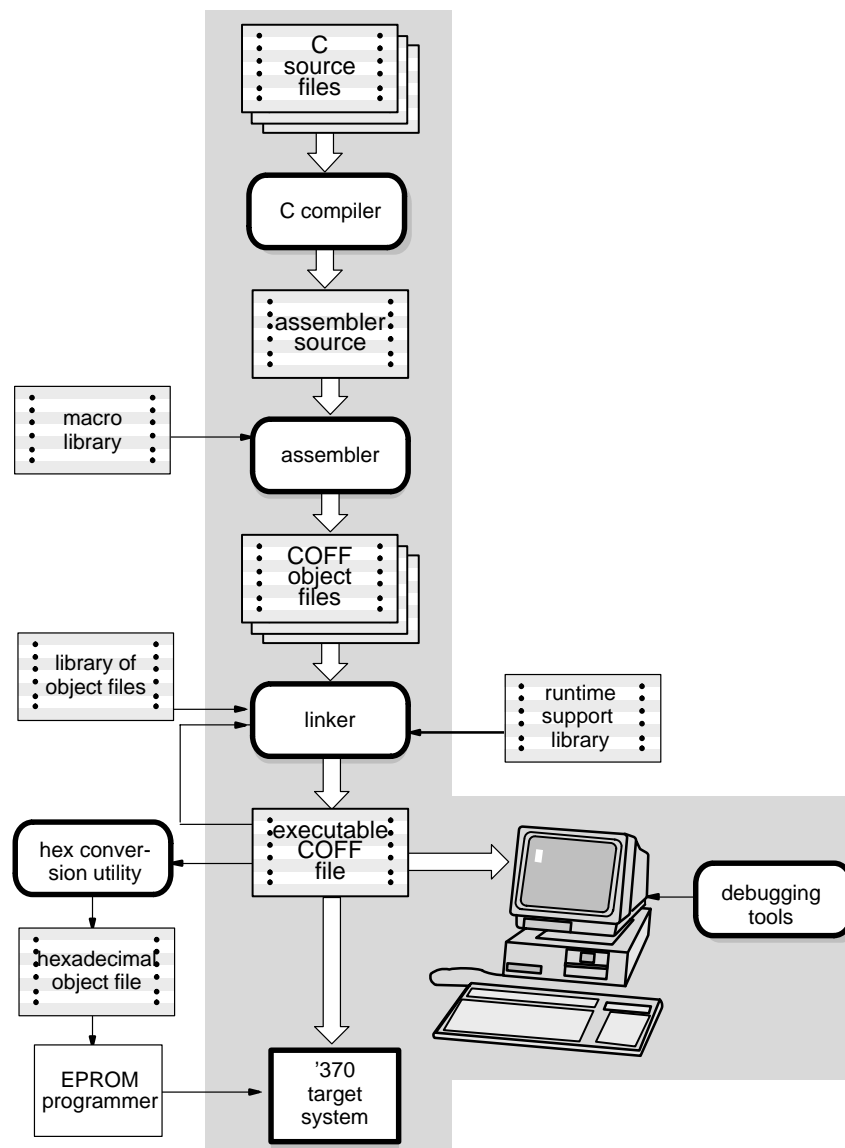
These tools use common object file format (COFF), which encourages modular programming. COFF allows you to divide your code into logical blocks, define your system's memory map, and then link code into specific memory areas. COFF also provides rich support for source-level debugging.

The following list describes the tools shown in Figure 1–3.

C compiler

The '370 **optimizing ANSI C compiler** is a full-featured optimizing compiler that translates standard ANSI C programs into '370 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.

❏ *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, 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 '370 assembly language source files into machine language object files.

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 **'370 target system.** You can use one of several **debugging tools** to refine and correct your code. Available products include:

❑ A realtime in-circuit **emulator**, and

❑ An **application board.**

hex conversion utility

A **hex conversion utility** is also available; it converts a COFF object file into an ASCII-Hex, Intel, Motorola-S, Tektronix, or TI-tagged object-format file that can be downloaded to an EPROM programmer.

1.4 Preparing Your Program for Debugging

Figure 1–4 illustrates the steps you must go through to prepare a program for debugging.

Figure 1–4. Steps You Go Through to Prepare a Program
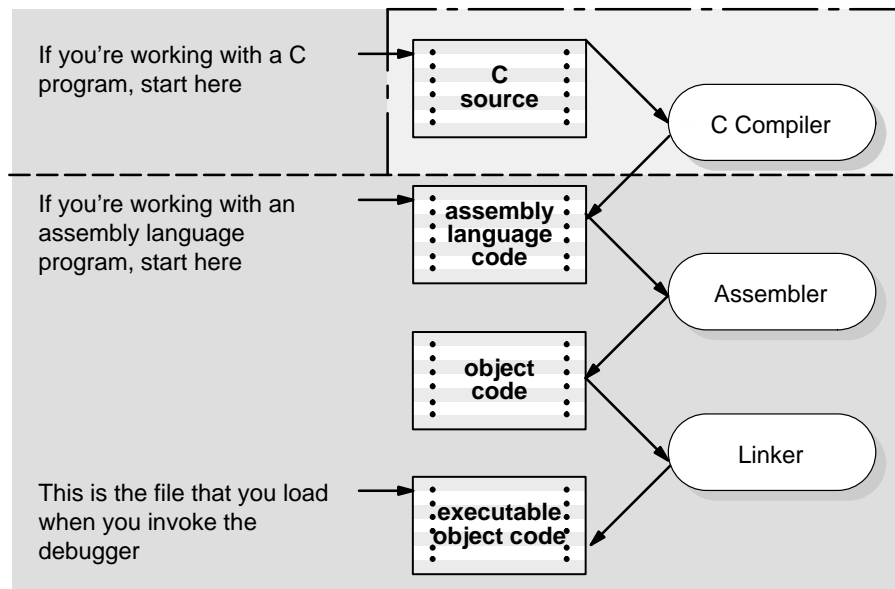
If you're preparing to debug a C program. . .	1) Compile the program; **use the –g option.** If you plan to use the Profiler, compile the program with the –as option.
	2) Assemble the resulting assembly language program.
	3) Link the resulting object file.
	This produces an object file that you can load into the debugger.
If you're preparing to debug an assembly language program. . .	1) Assemble the assembly language source file. If you want to include a local symbol table, assemble the program with the –s option.
	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 cl370 shell program. The *TMS370 Assembly Language Tools User's Guide* and *TMS370 C Compiler User's Guide* contain complete instructions for invoking the tools individually and for using the shell program.

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

> **cl370**   [−*options*]   **−g**   [*filenames*]   [**−z** [*link options*]]

cl370   is the command that invokes the compiler and assembler.

options   affect the way the compiler processes input files. If you plan to use the debugger in a profiling environment, include the –as option.

−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.

filenames   are one or more C source files, assembly language source files, or object files. Filenames are not case sensitive.

−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	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

1.5   **Invoking the Debugger**

Here's the basic format for the commands that invoke the debugger:

emulator:	**xds370**	[*filename*]	[*–options*]
application board:	**abd370**	[*filename*]	[*–options*]

xds370

and **abd370**   are the commands that invoke the debugger for each tool. If you are using Microsoft Windows, use **xds370w** to invoke 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 path-name. 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 (Table 1–1 summarizes the available options).

You can also specify filename and option information with the D_OPTIONS environment variable (see *Setting up the environment variables* in the appropriate installation guide). Table 1–1 lists the debugger options and specifies which debugger tools use the options; the subsections following describe the options.

Table 1–1. Summary of Debugger Options

Option	Brief description	Debugger Tools
–b[b]	Select the screen size	All
–i *pathname*	Identify additional directories	All
–p *serial port*	Identify the serial port	Emulator
–profile	Enter the profiling environment	Emulator
–s	Load the symbol table only	All
–t *filename*	Identify a new initialization file	All
–v	Load without the symbol table	All
–x	Ignore D_OPTIONS	All

Selecting the screen size (–b option)

By default, the debugger uses an 80-character-by-25-line screen. You can use one of the options in Table 1–2 to specify a different screen size.

Table 1–2. Screen Size Options (for Use With the –b Option)

Option	Description	Notes
none	80 characters by 25 lines	Default display
–b	80 characters by 43 lines	Any EGA or VGA display
–bb	80 characters by 50 lines	VGA only

Note:

Using the –b options override the *init.clr* file.

Identifying additional directories (–i option)

The –i option 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. For example:

xds370 –i $path_1$ **–i** $path_2$ **–i** $path_3$ . . .

Using –i is similar to using the D_SRC environment variable (see *Setting up the environment variables* in the appropriate chapter of your installation book). 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 –i, D_SRC, and the debugger USE command).

Identifying the serial port (–p option)

The –p option is valid only when using the emulator. The –p option identifies the serial port that the debugger uses for communicating with the emulator or the application board. The default setting, –p1, is used when your serial port is connected to COM1. Depending on your serial port connection, replace *port address* with one of these values:

❏ If you are using serial communication port 1, enter:

```
xds370 –p1
```

❏ If you are using serial communication port 2, enter:

```
xds370 –p2
```

If you use the wrong setting, you'll see this error message when you try to invoke the debugger:

```
 CANNOT INITIALIZE TARGET SYSTEM ! !
– Check communications port selection
  – Check cabling and target power
```

Entering the profiling environment (–profile option)

This option is valid only when you are using the emulator. The –profile option allows you to bring up the debugger in a profiling environment so you can collect statistics about code execution. Note that only a subset of the base debugger features is available in the profiling environment.

Loading the symbol table only (–s option)

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.

Identifying a new initialization file (–t option)

The –t option allows you to specify an initialization command file that will be used instead of init.cmd. If –t is present on the command line, the file specified by *filename* will be invoked as the command file instead of init.cmd.

Loading without the symbol table (–v option)

The –v 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 those performed when you invoke the debugger and those performed with the LOAD command within the debugger environment.

Ignoring D_OPTIONS (–x option)

The –x option tells the debugger to ignore any information supplied with D_OPTIONS. For more information about D_OPTIONS, refer to the *TMS370 Family XDS/11 Installation Guide,* the *TMS370 Family XDS/22 Installation Guide,* or the *TMS370 Family Application Board Installation Guide.*

1.6 Exiting the Debugger

To exit any version of 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.

If you are running the debugger under Microsoft Windows, you can also exit the debugger by selecting the exit option from the Microsoft Windows menu bar.

1.7   Debugging '370 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 1.4, *Preparing a Program for Debugging*, page 1-11.

Step 2

Ensure that the debugger has a valid memory map.

See Chapter 6, *Defining a Memory Map.*

Step 3

Load the program's object file.

See Section 7.3, *Loading Object Code*, page 7-10.

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 7-12.

Step 5

Stop the program at critical points and examine important information.

See Chapter 9, *Using Breakpoints,* and Chapter 8, *Managing Data.*

Step 6

If you find minor problems in your code, you can temporarily solve them with patch assembly.

See *Modifying assembly language code* on page 7-5.

Step 7

Once you have decided what changes must be made to your program, exit the debugger, edit your source file, and return to step 1.

An Introductory Tutorial

This chapter provides a step-by-step demonstration of the '370 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 that they're presented. This tutorial contains two sets of lessons (11 in the first, 13 in the second) and takes about one hour to complete.

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. . .

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

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.

Important! This step assumes that you are using the default serial port or that you have identified the serial port with the D_OPTIONS environment variable, as described in the *Identifying the serial port (–p option)*, on page 1-14.

Invoke the debugger and load the sample program:

❑ For the **emulator**, enter:

    `xds370 c:\370tools\sample` ⏎

❑ For the **application board**, enter:

    `abd370 c:\370tools\sample` ⏎

Take a look at the display. . .

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

COMMAND window display area →

memory contents →

command line →

```
 Load    Break    Watch    Memory    Color    MoDe    BTT    Run=F5    Step=F8    Next=F10
 ┌─DISASSEMBLY ───────────────────────────────────────────┐      ┌─CPU────────────┐
 7185  88          c_into0:   MOVW    #04883h,R021                   PC     7185
 7189  98                     MOVW    R021,R01F                      A      87
 718c  52                     MOV     #022h,B                        B      00
 718e  fd                     LDSP                                   ST     40
 718f  8e                     CALL    7199h                          SP     22
 7192  8e                     CALL    main
 7195  8e                     CALL    exit
 7198  fa                     RTI
 7199  88                     MOVW    #0723Ah,R0F
 719d  00                     JMP     71BFh
 719f  f4                     MOV     3(R0F),A
 71a3  d0                     MOV     A,R0D
 71a5  f4                     MOV     2(R0F),A
 71a9  d0                     MOV     A,R0C
 71ab  70                     INCW    #4,R0F
 71ae  00                     JMP     71BAh

 ┌─COMMAND ──────────────────────┐
 (c)Copyright 1992, Texas Instruments     ┌─MEMORY────────────────────────────────┐
 Silicon Revision 2
 Emulator Revision 1               0000 87 00 cb 01 00 00 28 e5 00 00 00 00
 Loading sample.out                000c 28 81 72 44 00 00 00 00 00 00 00 00
 Done                              0018 00 00 00 00 00 00 28 87 28 8f 00 70
 _                                 0024 40 00 00 00 00 00 00 00 00 00 00 00
 >>>                               0030 00 00 00 00 00 00 00 00 00 00 00 00
```

❑ If you **don't** see a display, then your debugger or system may not be installed properly. Go back through the installation instructions and be sure that you followed each step correctly; then reinvoke the debugger.

❑ 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 JMP instructions or say* Invalid address*—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 '370 processor:

reset ⏎

2) Load the sample program again:

load c:\370tools\sample ⏎

❑ If you see a display and the first few lines of the DISASSEMBLY window still show JMP instructions or they say invalid address after resetting the '370 processor, your cord (RS232) may not be inserted snugly. Check your cord and see if it is installed correctly, then reenter the above commands.

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 0x7189.

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

mem 0x7189   🖉

Notice that the first column in the DISASSEMBLY window corresponds to the addresses in the MEMORY window.

Try This: The highlighted statement in the DISASSEMBLY window shows that the PC is currently pointing to address 0x7189. You can modify the MEMORY display 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 at a time can be active.

Make the CPU window the active window:

win CPU 🖉

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.

Important! If you don't see a change in the appearance of the CPU window, 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 parameters in uppercase as shown.

Try This: Press the F6 key to "cycle" 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.

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 the 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.

2-6

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.

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 process forces 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 $\boxed{\it{\approx}}$

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.

"Unzoom" or return the window to its previous size by entering the ZOOM command again:

zoom $\boxed{\it{\approx}}$

The ZOOM command will be recognized even though the COMMAND window is hidden by the CPU window.

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 and on the position of the window before you tried to move it.

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 process forces 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 the up or down scroll arrow.

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 13.5 (page 13-58) 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 become 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 codes of the first eleven statements in the DISASSEMBLY window are highlighted; this is because these statements are associated with the current C statement (line 55 in the FILE window).

❏ The CALLS window, which tracks functions as they're called, now points to main().

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

Become familiar with the three debugging modes

The debugger has three basic debugging modes:

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

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

❏ **Assembly mode** shows only the disassembly, no C, 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) Press 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 click the left mouse button.

You should be in auto mode now, and you should see the FILE window but not the DISASSEMBLY window (because you're 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.
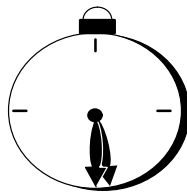
Halfway Point

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

To close the debugger, just type **QUIT** ⟦⤶⟧. When you return to the debugger, you must reinvoke it and load the sample program (refer to page 2-3). Turn to page 2-14 and continue with the second set of lessons.

Open another text file, then redisplay a C source file

In addition to what you already know about the FILE window and the FILE com-
mand, 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 dis-
   played in the FILE window).

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

file init.cmd 　　　🔎

This replaces sample.c in the FILE window with the initialization batch file
(init.cmd) 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 $\boxed{\text{⏎}}$

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

This isn't very exciting: halt program execution:

$\boxed{\text{ESC}}$

Set some software 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 $\boxed{\text{ESC}}$, 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 *software breakpoints*.

Here's an example of the debugger's informative capabilities (more are coming). You're going to benchmark some 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.

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 $\boxed{\text{⏎}}$

Set some software breakpoints:

1) Scroll to line 60 in the FILE window (the meminit() statement) and set a software breakpoint at that line:

   ↖ a) Point the mouse cursor at the statement on line 60.

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

2) Set another software breakpoint at line 68 (the for (;;); statement).

3) Reset the program entry point:

   restart 🔎

4) Enter the run command:

   run 🔎   *This runs to the breakpoint*

Benchmark some code:

1) Enter the runb command:

   runb   *This runs to the second breakpoint*

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

   ? clk

XDS/22 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 break-pointed 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 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 the 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 sp ⏎
wa pc, Program Counter ⏎
wa *0x4000, 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.

Resize the WATCH window if it isn't wide enough to display the PC value. You need to single-step through the loop that you benchmarked in the previous step.

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 ⏎

Observe the FILE, DISASSEMBLY, and WATCH windows.

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

Try executing this loop one more time. Take a look at this code; it's 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 software 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 $\boxed{\nearrow}$
wd 2 $\boxed{\nearrow}$
wd 1 $\boxed{\nearrow}$

Set up for the conditional run examples:

1) Set software breakpoints at lines 57 and 66.

2) Set up for conditional run example:

restart $\boxed{\nearrow}$

run $\boxed{\nearrow}$

3) Initiate the conditional run:

run i<100 $\boxed{\nearrow}$

This causes the debugger to run through the 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 $\boxed{\nearrow}$

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 declared in the sample program:

```
whatis genum  🠪
      enum yyy genum;              genum is an enumerated type
whatis tiny6  🠪
      struct {                              tiny6 is a structure
          int u;
          int v;
          int x;
          int y;
          int z;
      } tiny6;
whatis call  🠪
      int call();       call is a function that returns an integer
whatis s  🠪
      short s;              s is a short unsigned integer
whatis zzz  🠪
      struct zzz {              zzz is a very long structure
          int b1;
          int b2;
```
Press `ESC` to halt long listings

Clear the COMMAND window display area

After displaying all of these types, you may want to clear them away. This is easy to do.

Clear the COMMAND window display area:

```
cls  🠪
```

Try This: CLS isn't the only system-type command that the debugger supports.

```
cd ..                    Change back to the main directory
dir                      Show a listing of the current directory
cd directory name        Change back to the debugger 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 a structure in a DISP window:

disp tiny6 ⟳
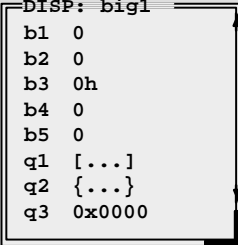
Close the DISP window:

F4

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: big1 ═
  b1  0
  b2  0
  b3  0h
  b4  0
  b5  0
  q1  [...]
  q2  {...}
  q3  0x0000
```

❑ 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 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, named `big1.q3`, that shows what q3 is pointing to (it's pointing to another structure). Close this DISP window or move it out of the way.

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.)

key

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 .

Close all of the DISP windows:

1) Make big1's DISP window 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 0x4000 as type float (exponential floating-point format).

Display memory contents in floating-point format:

disp *(float *)0x4000 ⏎

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 0x4000—*it isn't memory location 0*. Note that you can scroll through the memory displayed in the DISP window; item [1] is at 0x4003, and item [–1] is at 0x3FFd.

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 i ⏎   *Type int*
    wa f ⏎   *Type float*

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, f   ⏎                      Floats as octal 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 i, NEWi  ⏎
wa f, NEWf  ⏎
```

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:

```
? i,c   ⏎
```

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

```
wa str.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 0x4000:

        mem 0x4000   ⏎

    2) Point to the contents of memory location 0x4000.

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

    4) Type 00.

    5) Press ⏎ to enter the new value.

    6) Press ESC to conclude editing.

Try This: Here's another method for editing data that 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 7000.

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

    6) Type 99.

⏎ 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*. When you invoked the debugger for this tutorial, the debugger automatically read a default memory map from a batch file included in the 370tools directory. For the purposes of the sample program, that's fine (which is why this lesson was saved for next-to-last).

View the default memory map settings:

ml ⏎

Look in the COMMAND window display area—you'll see a listing of the areas that are currently mapped.

It's easy to add new ranges to the map or delete existing ranges.

Change the memory map:

1) Use the MD (memory delete) command to delete a block of memory:

    md 0x4000 ⏎

    This deletes the block of memory beginning at address 0x2000.

2) Use the MA (memory add) command to define a new block of memory:

    ma 0x4000,0xfff,RAM ⏎

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 short-hand 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 0x4000,0xfff,RAM;ml" [⏎]

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

   mymap [⏎]

   This is equivalent to entering the following three commands:

   ```
   mr
   ma 0x4000,0xfff,RAM
   ml
   ```

Close the debugger

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

Close the debugger and return to the operating system:

quit [⏎]

Tutorial: Using BTT Features

This chapter provides a step-by-step, hands-on demonstration of basic BTT (breakpoint, trace, and timing) features. This tutorial takes about one hour to complete.

This tutorial can be used only with the XDS/22 emulation system.

Understanding the example program

The tutorial uses one program, named *example*, to illustrate trace analyzer capabilities. This program is not intended to represent a real program—it is provided for illustration purposes only. The lessons in this tutorial will be more useful if you have an understanding of the example program.

The example program generates random numbers in the range 1–256. It writes the random numbers to an array named *results.* The variable *rnum* represents the current random number. The array *previous* is an index into the results array. The two members of previous are set to the values of the two previous random numbers so that the current random number is stored at results[previous random number]. The program calls two assembly language functions, *pinhi* and *pinlow*.

Figure 3–1 *(a)* shows the C portion of the example program, and Figure 3–1 *(b)* shows the assembly language portion. Table 3–1 lists the first 48 numbers that are generated by the example program (some of the lessons use specific data points, so it may be useful to know where they come from).

Figure 3–1. Example Program for the Tutorial

(a) example.c

```
#define SEED 1
extern ioinit();
extern pinhi();
extern pinlow();
int results[256];                   /* data storage area */
int previous[2] = {0, 1};           /* previous 2 random numbers */
int rnum = SEED;                    /* random number */
main()
{
   ioinit();
   for ( ; ; )
   {
       previous[0] = previous[1];  /* save the two previous random numbers */
       previous[1] = rnum;
       random(&rnum);  /* generate a random number */
       /*********************************************/
       /* switch on the two least significant bits of */
       /* previous random number                   */
       /*********************************************/
       switch(rnum & 3)
       {
           case 0 :  write_number();
                     break;
           case 1 :  write_number();
                     break;
           case 2 :  write_number();
                     pinhi();
                     break;
```

Figure 3–1. Example Program for the Tutorial (Continued)

(a) example.c (continued)

```
            case 3 :  write_number();
                      pinlow();
                      break;
        }
    }
}
write_number()
{
    results[previous[0]] = rnum;
}
random(r)
int *r;
{
    *r = rand() & 0xFF;
}
```

(b) externs.asm

```
            .global   _ioinit, _pinhi, _pinlow
_ioinit     mov       #1, P023
            rets
_pinhi      mov       #1, P022
            rets
_pinlow     mov       #0, P022
            rets
            .end
```

Table 3–1. First 48 Numbers Generated by the Example Program

65	(41h)	35	(23h)	44	(2Ch)	49	(31h)
22	(16h)	47	(2Fh)	117	(75h)	45	(2Dh)
39	(27h)	125	(7Dh)	66	(42h)	25	(19h)
68	(44h)	68	(44h)	51	(33h)	105	(69h)
121	(79h)	98	(62h)	77	(4Dh)	53	(35h)
21	(15h)	82	(52h)	100	(64h)	59	(3Bh)
89	(59h)	98	(62h)	119	(77h)	125	(7Dh)
28	(1Ch)	99	(63h)	111	(6Fh)	16	(10h)
63	(3Fh)	105	(69h)	110	(6Eh)	27	(1Bh)
15	(Fh)	19	(13h)	86	(56h)	122	(7Ah)
10	(Ah)	80	(50h)	97	(61h)	94	(5Eh)
49	(31h)	40	(28h)	74	(4Ah)	52	(34h)

Invoke the debugger and load the example program

Included with the debugger is a BTT demonstration program named *example*. This lesson shows you how to invoke the debugger. Use the –b option so that the debugger uses a larger display.

Invoke the debugger:

❏ If you are using serial communication port 1, enter:

xds370 example [⏎]

❏ If you are using serial communication port 2, enter:

xds370 –p2 example [⏎]

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

```
 Load    Break    Watch    Memory    Color    MoDe    BTT    Run=F5    Step=F8    Next=F10
┌─DISASSEMBLY ─────────────────────────────────────────────┐        ┌─CPU──────────┐
 71b8  88          c_int00:   MOVW    #0210Ah, R021                    PC     71b8
 71bc  98                     MOVW    R021, R01F                       A      0e
 71bf  52                     MOV     #022h, B                         B      00
 71c1  fd                     LDSP                                     ST     a0
 71c2  8e                     CALL    71CCh                            SP     24
 71c5  8e                     CALL    main
 71c8  8e                     CALL    exit
 71cb  fa                     RTI
 71cc  88                     MOVW    #0726Ch, R0F
 71d0  00                     JMP     71F2h
 71d2  84                     MOV     3(R0F), A

└──────────────────────────────────────────────────────────┘

┌═COMMAND ═══════════════════════┐
 370 XDS v2.06 BTT v1.4                      ┌─MEMORY──────────────────────────────────┐
                                              0000 03 00 70 46 db 68 20 02 7f 75 4b 81
 Loading example.out                          000c 21 08 72 84 00 00 00 00 00 00 00 00
    39 Symbols loaded                         0018 00 00 00 00 00 00 21 0e 21 0e 00 70
 Done                                         0024 53 00 00 00 00 00 00 00 00 00 00 00
 ─                                            0030 00 00 00 00 00 00 00 00 00 00 00 00
 >>> ▮                                       └─────────────────────────────────────────┘
└────────────────────────────────┘
```

Open the BTT setup dialog box

Many of the BTT settings that you must define are made from within the BTT Setup dialog box. In this lesson, you'll learn how to open this dialog box.

Use the BTT Setup dialog box:

1) Open the BTT Setup dialog box:

   BTT→Setup

   The dialog box is labeled like a window. Beneath the BTT Setup *label, you should see another label that says* State 0. *That means that this is where you can define and view information related to state 0.*

2) Click on the <Next state> field.

   The label now says State 1. *There are a total of four states, labeled state 0–state 3. For each state, you can define actions such as traces and hardware breakpoints.*

3) Click on <Next state> until you're back to state 0.

4) Click on <Cancel> to close the BTT Setup dialog box.

Collect traces on a specific address

In this lesson, you'll:

❏ Learn how to select the state mode,
❏ Define conditions for collecting trace samples (the example program tem-
   porarily stores random numbers into an array named previous; you'll col-
   lect trace samples whenever previous[1] is accessed),
❏ Learn how to open the INSPECT window,
❏ Run the example program (collecting samples during the run), and
❏ Halt program execution by pressing ESC.

Set up for tracing:

1) Open the BTT Setup dialog box:

   BTT→Setup

2) Make sure that you're in state 0.

3) In the State mode portion of the dialog box, click on ( )Address only.

   This selects address-only state mode.

Selecting this state mode provides you with additional BTT resources. The default state mode (address-and-data state mode) limits some of the BTT resources, so if you don't plan to use data values for defining traces, break-points, etc., you should select address-only state mode to ensure that you have access to all available resources.

---

4) Define conditions for a trace:

   a) In the BTT Setup dialog box, click on <Add action...>.

   The debugger displays the Select action menu.

   b) Click on <Trace>.

   This opens a dialog box where you can define the conditions for a trace action.

   c) In the Address qualifiers box, (*)One point—which is the default selection—should already be selected. If it isn't selected, click on it.

   d) In the addr1 field, type:

   previous+1

   e) Click on <OK> to indicate that you are finished defining conditions for the trace.

   This closes the Trace action dialog box and returns you to the BTT Setup dialog box. Information is now displayed about the trace action that you defined; the address is shown as 4001, *which is the address of previous[1].*

5) Click on <OK> to indicate that you are finished setting up the BTT.

   This closes the BTT Setup dialog box.

6) Open the INSPECT window:

   BTT→Inspect

7) Begin running the program, then halt program execution:

   run ⏎

   [ESC]

The traces you collected are now displayed in the INSPECT window; the contents of this window will be examined after the next lesson.

Trace on a specific address; breakpoint on address and data

In the preceding lesson, you collected traces whenever the address of previous[1] was accessed. The number of traces you collected was random because program execution was halted by pressing ESC.

In this lesson, you'll:

❑ Halt tracing by setting a breakpoint to occur when the value 42 is written to previous[1].

❑ Switch back to address-and-data state mode because you'll be using a data value when you define the breakpoint conditions.

The INSPECT window, opened in the preceding lesson, remains open. The traces collected in the preceding lesson will be overwritten by the traces collected in this lesson.

Set up the breakpoint:

1) Open the BTT Setup dialog box:

   BTT→Setup

2) Make sure that you're in state 0.

3) In the State mode portion of the dialog box, click on ( )Address and data.

   This selects address-and-data state mode.

4) The trace conditions remain from the preceding lesson; it is not necessary to redefine or alter them.

5) Define conditions for the breakpoint:

   a) Click on <Add action...>.

      The debugger displays the Select action menu.

   b) Click on <BP/event>.

      The debugger opens a dialog box where you can define conditions for a BP/event action.

> c) In the Address qualifiers box, (*)One point should already be selected.
>
> d) In the addr1 field, type:
>
>   **previous+1**
>
> e) In the Data qualifiers box, (*)One point should already be selected.
>
> f) In the data1 field, type:
>
>   **42**
>
> g) In the Cycle qualifiers box, click on [X]MR and [X]IAQ so that only [X]MW is selected.
>
> h) Click on <OK>.
>
>   *This returns you to the BTT Setup dialog box.*
>
> 6) Click on <OK>.
>
>   *This closes the BTT Setup dialog box.*
>
> 7) Run the example program:
>
>   **rrun** ⏎

View the contents of the trace buffer

The traces collected in the preceding lesson are visible in the INSPECT window. You should see a window like the one below:

```
Inspect
INDX  ST   h   m   s ms   us   ns   EXTERNAL CYCLE  ADDR  DATA   REVERSE ASM
0000  0       0:00:00.000 057 400 11111111   WRITE  4001   01
0001  0       0:00:00.000 264 400 11111111   READ   4001   01
0002  0       0:00:00.000 271 200 11111111   WRITE  4001   01
0003  0       0:00:00.000 594 800 11111111   READ   4001   01
0004  0       0:00:00.000 601 600 11111111   WRITE  4001   41
0005  0       0:00:00.000 933 800 11111111   READ   4001   41

T1 0:00:00.000 000 000  AVG1 0:00:00.000 000 000  T2 0:00:00.000 000 000
```

The INSPECT window is divided into two parts. The last line shows timing statistics (these are discussed later in the tutorial). The upper portion shows the contents of the trace buffer. Each entry in the upper portion shows a single trace sample. Each trace sample shows:

❑ The position of the trace sample within the trace buffer (INDX field). In general, trace sample 0000 would be the first sample collected, and trace sample 2046 would be the last sample collected. However, you may not always collect a full 2047 samples.

❑ The state during which the trace sample was collected (ST field).

❑ Timing information that indicates when the trace sample was collected (h to ns fields).

❑ Information about the values on external signals (EXTERNAL field).

❑ The type of memory cycle that took place when the trace sample was collected (CYCLE field).

❑ The value that was on the address bus (ADDR field).

❑ The value that was on the data bus (DATA field).

❑ Any assembly language code associated with the trace sample (REVERSE ASM field).

Display a specific trace sample

The trace buffer can hold up to 2047 trace samples. You can resize or scroll the INSPECT window to view additional entries. You can also display specific samples.

In this lesson, you'll display specific trace samples according to:

❑ The sample's position within the trace buffer,
❑ Specific conditions that the sample meets, or
❑ Whether or not the sample also met BP/event conditions.

Locate a trace sample based on its position within the trace buffer:

❑ Find the 30th trace sample.

1) Select the Position entry from the BTT menu:
   BTT→Position

   This opens a dialog box where you can enter the trace sample number.

2) In the Sample Number field, type:

   30

3) Click on <OK>.

   The display in the INSPECT window changes so that trace sample #30 is displayed in the center of the window.

❑ Find the last trace sample.

1) Select the Position entry from the BTT menu:
   BTT→Position

2) Click on <Bottom>.

   The number in the Sample Number field changes to show the number of the last trace sample.

3) Click on <OK>.

   The display in the INSPECT window changes so that the last trace sample is displayed on the last line.

❏ The last two parts of this lesson use the BTT→Lookup selection. Lookup locates the *next* trace sample that meets certain conditions. Because you're already at the last sample, there is no next sample. Move back to the beginning of the trace buffer:

1) Select the Position entry from the BTT menu:

   BTT→Position

2) In the Sample Number field, type:

   0

3) Click on <OK>.

   The display in the INSPECT window changes so that the first trace sample is displayed.

❏ Look for a trace sample where the value 63 (0x3F) is read from previous[1]:

1) Select the Lookup entry from the BTT menu:

   BTT→Lookup

   This opens a dialog box where you can enter the conditions that you're looking for. The dialog box looks very similar to the dialog boxes where you define conditions for traces, BP/events, etc.

2) In the Address and data qualifiers boxes, (*)One point should already be selected.

3) In the addr1 field, type:

   previous+1

4) In the data1 field, type:

   0x3f

5) In the Cycle qualifiers box, click on [X]MW and [X]IAQ so that only [X]MR is selected.

6) Click on <OK>.

   The display in the INSPECT window changes to display the trace sample that meets the defined conditions.

❑ Look for a trace sample that was also an event:

1) Select the Lookup entry from the BTT menu:

BTT→Lookup

This opens the same dialog box used in the preceding example.

2) In the Flag field at the top of the dialog box, click on ( )Last.

3) In the addr1 and data1 fields and their associated mask fields, type:

0

The addr1 and data1 fields must be cleared because they re-tained the data from your preceding use of BTT→Lookup. The mask fields must also be cleared; the purpose of the mask values is explained in a later lesson.

4) In the Cycle qualifiers box, click on [X]MW and [X]IAQ to re-enable them.

5) Click on <OK>.

The display in the INSPECT window changes to display the BP/ event, which was the last event collected that also met trace conditions.

Change the timing format of trace samples

The h–ns fields in the INSPECT window show information about when a trace sample was collected. By default, these fields show the total amount of time that has elapsed from when tracing began to when a particular trace sample was collected.

In this lesson, you'll change the format of the trace-sample timing statistics to show one of three measurements:

❑ The first format shows the time difference between any trace sample and the previously collected trace sample.

❑ The second format shows the time difference between any trace sample and a specific sample within the trace buffer; the specific sample is selected by cursor position.

❑ The third format returns you to the default timing statistics.

Change the format of the trace sample timing statistics:

❏ Show the time difference between any trace sample and the preceding trace sample.

    1)   Select the Format entry from the BTT menu:

        BTT→**Format**

        This opens a dialog box where you can select the time format.

    2)   Click on ( )Delta.

    3)   Click on <OK>.

        The times displayed in the INSPECT window change to show the difference between adjacent samples.

❏ Select a trace sample, then show timing statistics as the difference from the selected sample.

    1)   First, select a trace sample that will serve as a marker. For this example, use trace sample #48. Select Position from the BTT menu:

        BTT→**Position**

    2)   In the Sample Number field, type:

        `48`

    3)   Click on <OK>.

        The INSPECT window changes to display sample #48; the cursor is positioned on this trace sample.

    4)   Select the Format entry from the BTT menu:

        BTT→**Format**

    5)   Click on ( )Mark.

    6)   Click on <OK>.

        The times displayed in the INSPECT window change to show the time difference from sample #48. Samples collected before sample #48 are shown as negative times; samples collected after sample #48 are shown as positive times.

❑ Switch back to the default timing format, which shows the amount of time elapsed since tracing began.

1) Select the Format entry from the BTT menu:
   BTT→Format

2) Click on ( )Absolute.

3) Click on <OK>.
   The times displayed in the INSPECT window return to the default format.

Trace on one of two address values; halt on program time out

In addition to defining actions based on an address or data access, you can define an action based on one of two addresses or data values being accessed.

In this lesson, you'll:

❑ Collect trace samples whenever pinhi or pinlow is accessed.

❑ Use the time-out timer, which limits program execution time by specifying the maximum amount of time that your program can run.

Define a trace that is qualified by either of two addresses:

1) Open the BTT Setup dialog box:
   BTT→Setup

2) Click on <Clear state>.
   This clears the actions previously defined for state 0.

3) Define the conditions for tracing:

   a) Click on <Add action...>.
      The debugger displays the Select action menu.

   b) Click on <Trace>.
      The debugger opens the Trace action dialog box.

   c) In the Address qualifiers box, click on ( )Two points.

d) In the addr1 field, type:

pinhi

e) In the addr2 field, type:

pinlow

f) Click on <OK>.

This returns you to the BTT Setup dialog box.

4) Assign a value to the time-out timer:

a) Click on <Globals> (at the bottom of the dialog box).

This opens the Globals dialog box.

b) In the Time out field, type:

0001

The resulting value should look like this:

[0001.000 000 000]

This defines a time-out value of 1 second.

c) Click on <OK>.

This returns you to the BTT Setup dialog box.

5) Click on <OK> .

This closes the BTT Setup dialog box.

6) Run the example program:

rrun ⏎

Program execution halts after 1 second. It is not always necessary (or even desirable) to reset before running a program; however, for the purposes of this tutorial, resetting before running ensures that you will obtain the correct lesson results.

Trace on a range of data

In addition to defining actions based on one or two specific address or data values, you can define actions based on an address' or data value's relation-ship to a range.

In this lesson, you'll:

❑ Trace on values within a range of data.
❑ Trace on values that are outside of the data range.

The time-out timer defined in the preceding lesson will remain in effect for this lesson.

❑ Trace on data values within a range:

1) Open the BTT Setup dialog box:
   BTT→Setup

2) The dialog box shows a description of the preceding trace condi-tions. Point to this description and click the left mouse button.
   This opens the Trace action dialog box so that you can edit the existing trace settings.

3) Define the conditions for tracing:

   a) In the Address qualifiers box, click on (*)One point.

   b) In the addr1 field, type:

      0x4001

      This is the address of previous[1].

   c) In the Data qualifiers box, click on ( )In range.

   d) In the data1 field, type:

      90

   e) In the data2 field, type:

      130

   f) Click on <OK>.

      This returns you to the BTT Setup dialog box.

---

4) Click on <OK>.

This closes the BTT Setup dialog box.

5) Run the example program:

rrun ⏎

❑ Trace on data values outside of the range:

1) Open the BTT Setup dialog box:

BTT→Setup

2) Point to the description of the existing trace conditions and click the left mouse button.

This opens the Trace action dialog box so that you can edit the existing trace settings.

3) Redefine the trace conditions:

a) In the Data qualifiers box, click on ( )Outside range.

b) Click on <OK>.

This returns you to the BTT Setup dialog box.

4) Click on <OK>.

This closes the BTT Setup dialog box.

5) Run the example program:

rrun ⏎

Collect trace samples after breakpointing

Sometimes it is desirable to collect trace samples after a BP/event is met. This is useful when you are interested in collecting information about the state the processor is in after some event occurs. To do this, you define the BP/event and the trace conditions, then set up a mechanism that allows the BTT to recognize the BP/event when it occurs, but delays the BTT from actually executing the breakpoint until a specified number of trace samples are collected.

In this lesson, you'll:

❏ Collect trace samples after the value 65 is written to previous[1].

❏ Use a component called the delay counter to tell the BTT how many trace samples it should collect before executing the breakpoint.

Collect traces after a BP/event is recognized:

1) Open the BTT Setup dialog box.

   BTT→Setup

2) Assign a value to the delay counter and reset the time-out timer:

   a) Click on <Globals> (at the bottom of the dialog box).

   This opens the Globals dialog box.

   b) In the Delay count field, type:

   `50`

   This tells the BTT to collect 50 trace samples after the BP/event conditions are met. The breakpoint will be executed after the trace samples are collected.

   c) In the Time out field, type:

   `0000`

   The resulting value should look like this:

   `[0000.000 000 000]`

   This disables the time-out timer.

   d) Click on <OK>.

   This returns you to the BTT Setup dialog box.

---

3) Click on <Clear state>.

This clears the actions previously defined for state 0 but doesn't affect the global settings.

4) Define the conditions for the breakpoint:

  a) Click on <Add action...>.

  The debugger displays the Select action menu.

  b) Click on <BP/event>.

  The debugger opens the BP/event action dialog box.

  c) In the Address qualifiers box, (*)One point should already be selected.

  d) In the addr1 field, type:

  `previous+1`

  e) In the Data qualifiers box, (*)One point should already be selected.

  f) In the data1 field, type:

  `65`

  g) Click on <OK>.

  This returns you to the BTT Setup dialog box.

5) Define the the conditions for the trace samples:

  a) Click on <Add action...>.

  The debugger displays the Select action menu.

  b) Click on <Trace>.

  The debugger opens the trace action dialog box.

  c) In the Address qualifiers box, (*)One point should already be selected.

  d) In the addr1 field, type:

  `previous+1`

---

e) In the Data qualifiers box, click on ( )In range.

f) In the data1 field, type:

100

g) In the data2 field, type:

200

h) In the Cycle qualifiers box, click on [X]MR and [X]IAQ so that only [X]MW is selected.

i) Click on <OK>.

This returns you to the BTT Setup dialog box.

6) Click on <OK>.

This closes the BTT Setup dialog box.

7) Run the example program:

rrun ⏎

Collect reads and writes associated with IAQ cycles

The BTT supports two modes for collecting trace samples. So far, you have used only the default trace mode, called normal mode. In normal mode, the only samples that are collected are those that meet the trace conditions you've defined. A second trace mode, TRIX (trace instruction extended) mode, includes any reads and writes associated with a qualifying IAQ cycle.

In this lesson, you'll:

❑ Collect traces to the first assembly language instruction associated with this line:

```
results[previous[0]] = rnum;
```

which appears in the write_number function. (The first disassembly address associated with this line is at 0x7087.)

❑ Collect samples twice: once with normal trace mode and a second time with TRIX trace mode.

❑ Use a global setting called max trace, which defines the total number of trace samples that will be collected before program execution halts.

Run the first trace session using normal trace mode:

1) Open the BTT Setup dialog box:

   BTT→Setup

2) Click on <Clear state>.

   This clears the actions previously defined for state 0.

3) Define a max trace value and reset the delay count:

   a) Click on <Globals>.

   b) In the Delay count field, type:

      0

      This disables the delay count.

   c) In the Max trace field, type:

      5

      The BTT will collect 5 trace samples before halting program execution.

   d) Click on <OK>.

      This returns you to the BTT Setup dialog box.

4) Define the conditions for tracing:

   a) Click on <Add action...>.

      The debugger displays the Select action menu.

   b) Click on <Trace>.

      The debugger opens the trace action dialog box.

   c) In the Address qualifiers box, (*)One point should already be selected.

   d) In the addr1 field, type:

      0x7087

   e) Click on <OK>.

      This returns you to the BTT Setup dialog box.

5) Click on <OK>.

This closes the BTT Setup dialog box.

6) Restart the program entry point and run the example program:

rrun ⏎

All the trace samples show this same information:

```
CYCLE ADDR   DATA   REVERSE ASM
IAQ   7087  8A     MOV.bss,A
```

Run the second trace session using TRIX trace mode:

1) Open the BTT Setup dialog box:

BTT→Setup

2) In the Trace mode box, click on ( )TRIX.

This selects TRIX trace mode.

3) Click on <OK>.

This closes the BTT Setup dialog box.

4) Restart the program entry point and run the example program:

rrun ⏎

Now the INSPECT window should show any reads or writes that are associated with the instruction at address 0x7087:

```
CYCLE ADDR   DATA   REVERSE ASM
IAQ   7087  8A     MOV.bss,A
READ  7088  20
READ  7089  00
READ  2000  01
WRITE 0000  01
```

Use a masked data value

When you define the conditions for an action, you can identify specific bits within an address or data value that should be ignored. To do this, you use a **mask**. A mask is a value that is ANDed with another value, resulting in the actual value to be used in the condition. The mask has 0s in the bit positions that should be ignored in the original data value or address.

In this lesson, you'll mask the data value and collect trace samples only when the last digit is a 5. Here's how the mask value is calculated:

Data value:	1 1 1 1	0 1 0 1	$F5_{16}$
Mask value:	0 0 0 0	1 1 1 1	$0F_{16}$
Qualifying values:	X X X X	0 1 0 1	
	don't care	5	

❑ The **data value** is specified as 0xF5:

■ The first hex digit doesn't matter, because it will be masked out; for convenience, specify it as F.

■ The second hex digit must be a 5 because you will be collecting values that end in 5.

❑ The **mask value** is specified as 0x0F:

■ The first hex digit must be a 0 because this is the don't-care part of the value.

■ The second hex digit must be F because F AND 5 produces 5.

Define trace conditions using a masked data value:

1) Open the BTT Setup dialog box:
   BTT→**Setup**

2) Click on <Clear state>.
   This clears the actions previously defined for state 0.

3) Define a larger max trace value:

   a) Click on <Globals>.

---

b) In the Max trace field, type:

2000

The BTT will collect 2000 trace samples before halting program execution.

c) Click on <OK>.

This returns you to the BTT Setup dialog box.

4) Define conditions for tracing:

a) Click on <Add action...>.

The debugger displays the Select action menu.

b) Click on <Trace>.

The debugger opens the trace action dialog box.

c) In the Address qualifiers box, (*)One point should already be selected.

d) In the addr1 field, type:

previous+1

e) In the Data qualifiers box, (*)One point should already be selected.

f) In the data1 field, type:

0xF5

g) In the mask field, type:

0x0F

h) Click on <OK>.

This returns you to the BTT Setup dialog box.

5) Click on <OK>.

This closes the BTT Setup dialog box.

6) Run the example program:

rrun ⏎

Examine the INSPECT window; all the values listed in the DATA field end with a 5.

Collect timing statistics

You can use two timers, the point timer and the range timer, for collecting program-timing statistics. The timers work similarly, but the method for defining conditions for them differs slightly.

This lesson is divided into two parts. In the first part, you'll:

❏ Assign a time-out value to limit program run time to two seconds. (Because you won't collect any traces in this lesson, the max trace value from the previous lesson would not be useful.)

❏ Define an action that starts the point timer whenever the pinhi function is accessed and stops the point timer when the pinlow function is accessed.

❏ Examine the collected timing statistics.

Note that you will not be collecting trace samples in either part of the lesson—it is not necessary to collect trace samples in order to gather timing statistics.

Define conditions for a point timer action:

1) Open the BTT Setup dialog box:

   BTT→**Setup**

2) Click on <Clear state>.

   This clears the actions previously defined for state 0.

3) Reset the max trace value and assign a time-out value:

   a) Click on <Globals>.

   b) In the Max trace field, type:

      `0`

      This resets the max trace value.

   c) In the Time out field, type:

      `0002`

      The resulting value should look like this:

      `[0002.000 000 000]`

      This defines a time-out value of 2 seconds.

   d) Click on <OK>.

      This returns you to the BTT Setup dialog box.

---

4) Define a point timer action:

   a) Click on <Add action...>.

   The debugger displays the Select action menu.

   b) Click on <Point Timer>.

   This displays an action dialog box where you can define the conditions for starting and stopping the point timer. Look at the Address qualifiers box—no qualifiers are selected. You **must** *define two address points for a point timer, so selections are unnecessary.*

   c) In the addr1 field, type:

   pinhi

   d) In the addr2 field, type:

   pinlow

   e) Click on <OK>.

   This returns you to the BTT Setup dialog box.

5) Click on <OK>.

   This closes the BTT Setup dialog box.

6) Run the example program:

   rrun 🖰

The INSPECT window should now show the timing information that was collected. The last line of the window should look something like this (your numbers may be different):

```
T1 0:00:00.090 675 800   AVG1 0:00:00.001 416 800   T2 0:00:00.000 000 000
```

Notice that the statistics are not labeled point timer or range timer—they're labeled T1 and T2, for timer 1 and timer 2. Whatever timer action you define first is reported as timer 1; in this case, the point timer statistics are shown under timer 1. The timer action you define second is reported as timer 2; in this case, no second timer action was defined, so there are no timer 2 statistics.

The statistics listed for AVG1 show the average amount of time that timer 1 (in this case, the point timer) was run.

In this part of the lesson, you'll:

❏ Use the same time-out value used in the first part of the lesson.

❏ Define a new action that starts the range timer whenever a data value in the range 0–100 is accessed and stops the range timer whenever a value in the range 101–200 is accessed.

❏ Examine the collected timing statistics.

Define conditions for a range timer action:

1) Open the BTT Setup dialog box:
   BTT→**Setup**

2) Click on <Clear state>.
   This clears the actions previously defined for state 0.

3) Define a Range Timer action:

   a) In the BTT Setup dialog box, click on <Add action...>.
      The debugger displays the Select action menu.

   b) Click on <Range timer>.
      This displays an action dialog box where you can define the conditions for starting the range timer.

   c) In the Data qualifiers box, click on ( )In range.

   d) In the data1 field, type:

      0

   e) In the data2 field, type:

      100

   f) Click on <OK>.
      This closes the first action dialog box and opens a second box where you can define the conditions for stopping the range timer.

   g) In the Data qualifiers box, click on ( )In range.

   h) In the data1 field, type:

      101

---

i) In the data2 field, type:

200

j) Click on <OK>.

This closes the second action dialog box and returns you to the BTT Setup dialog box.

4) Click on <OK>.

This closes the BTT Setup dialog box.

5) Run the example program:

rrun ⏎

The timer 1 statistics now shown in the INSPECT window are the statistics for the range timer action.

Jump to another state

Another of the actions that you can define is a jump to another state. This type of action is useful for handling special cases.

In this lesson:

❑ You'll collect traces on writes to previous[1] until pinhi is accessed.

❑ When pinhi is accessed, you'll jump from state 0 to state 1. You'll collect traces whenever pinhi is accessed, until pinlow is accessed.

❑ When pinlow is accessed, you'll jump back to state 0 and continue collecting traces on writes to previous[1].

Set up jump actions:

1) Open the BTT Setup dialog box.
    BTT→**Setup**

2) Click on <Clear state>.
    This clears the actions previously defined for state 0.

3) In the State mode box, click on ( )Address only.
    This selects address-only state mode.

4) Define the conditions for tracing:

   a) Click on <Add action...>.

     The debugger displays the Select action menu.

   b) Click on <Trace>.

     The debugger opens the trace action dialog box.

   c) In the Address qualifiers box, (*)One point should already be selected.

   d) In the addr1 field, type:

     previous+1

   e) In the Cycle qualifiers box, click on [X]MR and [X]IAQ so that only [X]MW is selected.

   f) Click on <OK>.

     This returns you to the BTT Setup dialog box.

5) Define the conditions for jumping:

   a) Click on <Add action...>.

     The debugger displays the Select action menu.

   b) Click on <Jump>.

     The debugger opens a dialog box where you can define the conditions for a jump action.

   c) In the Jump to.. field, type:

     1

   d) In the Address qualifiers box, (*)One point should already be selected.

   e) In the addr1 field, type:

     pinhi

   f) Click on <OK>.

     This returns you to the BTT Setup dialog box.

6) Click on <Next state>.

  This takes you to state 1.

7) Click on <Clear state>.

   This clears the actions previously defined for state 1.

8) In the State mode box, click on ( )Address only.

   This selects address-only state mode.

9) Define the conditions for the second trace:

   a) Click on <Add action...>.

      The debugger displays the Select action menu.

   b) Click on <Trace>.

      The debugger opens the trace action dialog box.

   c) In the Address qualifiers box, (*)One point should already be selected.

   d) In the addr1 field, type:

      pinhi

   e) Click on <OK>.

      This returns you to the BTT Setup dialog box.

10) Define the conditions for jumping back to state 0:

   a) Click on <Add action...>.

      The debugger displays the Select action menu.

   b) Click on <Jump>.

      The debugger opens the jump action dialog box.

   c) The Jump to... field should already contain a 0.

      0 is the default value. Since you want to go back to state 0 when the new jump conditions are met, it's not necessary to edit the Jump to... field.

   d) In the Address qualifiers box, (*)One point should already be selected.

   e) In the addr1 field, type:

      pinlow

   f) Click on <OK>.

11) Assign a max trace value:

a) Click on <Globals>.

b) In the Max trace field, type:

2047

c) In the Time out field, type:

0000

The resulting value should look like this:

[0000.000 000 000]

This disables the time-out timer.

d) Click on <OK>.

This returns you to the BTT Setup dialog box. Note that although you are defining this value in state 1, it applies to all states.

12) Click on <OK>.

This closes the BTT Setup dialog box.

13) Run the example program:

rrun ⏎

Scroll through the INSPECT window and watch the ST field; you'll see that it switches from 0 to 1 and back again because traces were collected in both states.

Close the INSPECT window

This is the end of the tutorial.

Close the INSPECT window:

1) Make the INSPECT window the active window:

win INSPECT

2) Press F4 .

This closes the window.

Chapter 4

The Debugger Display

The '370 C source debugger has a window-oriented display. This chapter shows what windows can look like and describes the basic types of windows you will use.

4.1 Debugging Modes and Default Displays

The basic debugger environment 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.

These modes cannot be used within the profiling environment; only the COMMAND, PROFILE, DISASSEMBLY, and FILE windows are available.

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 4–1. 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 4–1. The DISASSEMBLY window displays the reverse assembly of memory contents.

Figure 4–1. Typical Assembly Display (for Auto Mode and Assembly Mode)


```
┌─────┬──────┬───────┬────────┬───────┬──────┬─────┬────────┬────────┬─────────┐
│Load │ Break│ Watch │ Memory │ Color │ MoDe │ BTT │ Run=F5 │ Step=F8│ Next=F10│
└─────┴──────┴───────┴────────┴───────┴──────┴─────┴────────┴────────┴─────────┘
 ┌─DISASSEMBLY────────────────────────────────────┐      ┌─CPU──────┐
 │                                                │      │          │
 │ 7185  88    c_int00: MOVW   #02883h,R021       │      │ PC  7185 │
 │ 7189  98             MOVW   R021,R01F          │      │ A    87  │
 │ 718c  52             MOV    #022h,B            │      │ B    00  │
 │ 718e  fd             LDSP                      │      │ ST   40  │
 │ 718f  8e             CALL   7199h              │      │ SP   22  │
 │ 7192  8e             CALL   main               │      │          │
 │ 7195  8e             CALL   exit               │      │          │
 │ 7198  fa             RTI                       │      └──────────┘
 │ 7199  88             MOVW   #0723Ah,R0F        │
 │ 719d  00             JMP    71BFh              │
 │ 719f  f4             MOV    3(R0F),A           │
 │ 71a3  d0             MOV    A,R0D              │
 │ 71a5  f4             MOV    2(R0F),A           │
 │ 71a9  d0             MOV    A,R0C              │
 │ 71ab  70             INCW   #4,R0F             │
 │ 71ae  00             JMP    71BAh              │
 │ 71b0  9a             MOV    @R0F,A             │
 └────────────────────────────────────────────────┘
 ┌─COMMAND──────────────────────┐  ┌─MEMORY────────────────────────────────────┐
 │ '370 Debugger Version 4.01   │  │                                            │
 │ Copyright (c) 1990, 1992 Texas Instr│ 0000 87 00 cb 01 00 00 28 e5 00 00 00 00│
 │ TMS370 Silicon Revision B    │  │ 000c 28 81 72 44 00 00 00 00 00 00 00 00  │
 │ Loading sample.out           │  │ 0018 00 00 00 00 00 00 28 87 28 8f 00 70  │
 │ _  67 symbols loaded         │  │ 0024 40 00 00 00 00 00 00 00 00 00 00 00  │
 │                              │  │ 0030 00 00 00 00 00 00 00 00 00 00 00 00  │
 │ >>>                          │  │                                            │
 └──────────────────────────────┘  └────────────────────────────────────────────┘
```

❑ When the debugger is running C code, you'll see a C display similar to the one in Figure 4–2. (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 4–2. Typical C Display (for Auto Mode Only)

```
┌────────────────────────────────────────────────────────────────────────────────────┐
│ Load   Break   Watch   Memory   Color   MoDe   BTT   Run=F5   Step=F8   Next=F10     │
├─ FILE: sample.c ─────────────────────────────────────────────────────────────────── │
│  0052      extern call();                                                         ▲  │
│  0053      extern meminit();                                                         │
│  0054                                                                                │
│  0055      main();                                                                   │
│  0056         {                                                                      │
│  0057            int i = 0;                                                          │
│  0058            int j = 0, k = 0;                                                   │
│  0059                                                                                │
│  0060            meminit();                                                          │
│  0061            for (i=0;i<0x70;i++)                                                │
│  0062              {                                                                 │
│  0063                 call(i);                                                       │
│  0064                 if (i & 1) j += i;                                             │
│  0065                 aai[k][k] = j;                                              ▼  │
│  0066                 if (!(i & 0xFFFF)) k++;                                         │
│  0067              }                                                                 │
├─ COMMAND ─────────────────────────────────────┬─ CALLS ──────────────────────────── │
│  TMS370 Silicon Revision B                  ▲ │ 1: main()                            │
│  Loading sample.out                           │                                      │
│    67 symbols loaded                          │                                      │
│  Done                                         │                                      │
│ ─file sample.c                              ▼ │                                      │
│ >>>█                                           │                                      │
└────────────────────────────────────────────────────────────────────────────────────┘
```
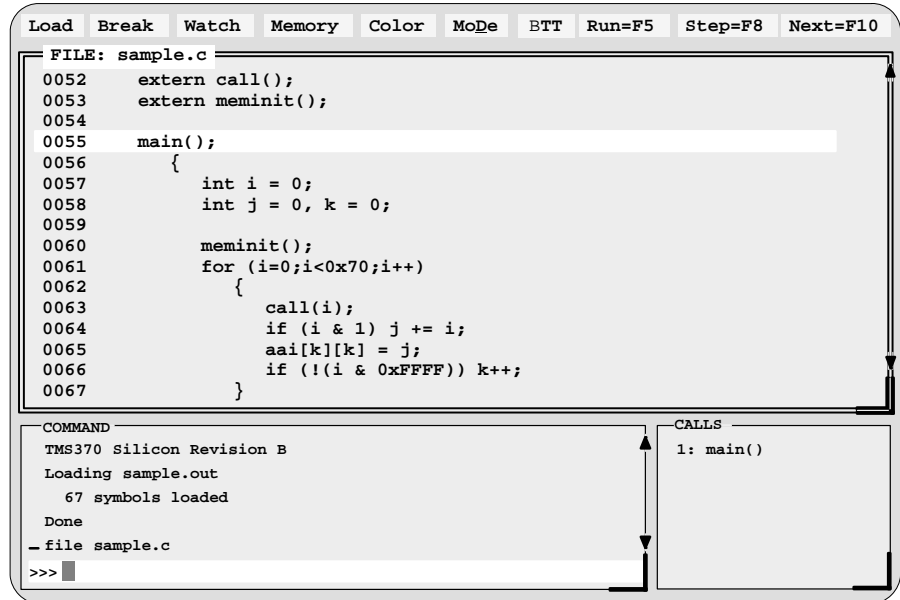
When you're running assembly language code, the debugger automatically displays windows as described for assembly mode.

When you're running C code, the debugger automatically displays the COMMAND, CALLS, and FILE windows. If you want, you can also open the INSPECT window, a WATCH window, and DISP windows.

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 4–1. 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 choose, you can also open the INSPECT window and a WATCH window in assembly mode.

Mixed mode

Mixed mode is for viewing assembly language and C code at the same time. Figure 4–3 shows the default display for mixed mode.

Figure 4–3. Typical Mixed Display (for Mixed Mode Only)

```
 Load   Break   Watch   Memory   Color   MoDe   BTT   Run=F5   Step=F8   Next=F10
 ┌─DISASSEMBLY─────────────────────────────────────────┐  ┌─CPU───────────────┐
 │                                                      │  │                   │
 │  7185   88     c_int00:   MOVW   #02883h,R021        │  │   PC  7185        │
 │  7189   98                MOVW   R021,R01F           │  │   A   87          │
 │  718c   52                MOV    #022h,B             │  │   B   00          │
 │  718e   fd                LDSP                       │  │   ST  40          │
 │  718f   8e                CALL   7199h               │  │   SP  22          │
 │  7192   8e                CALL   main                │  │                   │
 │  7195   8e                CALL   exit                │  │                   │
 │  7198   fa                RTI                        │  │                   │
 │  7199   88                MOVW   #0723Ah,R0F         │  └───────────────────┘
 │  719d   00                JMP    71BFh               │
 │  719f   f4                MOV    3(R0F),A            │
 │  71a3   d0                MOV    A,R0D               │
 ┌─FILE: sample.c──────────────────────────────────────┐  ┌─CALLS─────────────┐
 │  0052        extern call();                          │  │  1: main()        │
 │  0053        extern meminit();                       │  │                   │
 │  0054                                                │  │                   │
 │  0055        main();                                 │  │                   │
 │  0056        {                                       │  │                   │
 │  0057             int i = 0;                          │  │                   │
 │  0058             int j = 0, k = 0;                   │  │                   │
 │  0059                                                │  │                   │
 │  0060             meminit();                          │  │                   │
 │  0061             for (i=0;i<0x70;i++)                │  │                   │
 ┌─COMMAND─────────────────────────────┐  ┌─MEMORY──────────────────────────────┐
 │  Loading sample.out                 │  │  000087 00 cb 01 00 00 28 e5 00 00 00 00  │
 │                                     │  │  000c28 81 72 44 00 00 00 00 00 00 00 00  │
 │  Done                               │  │  001800 00 00 00 00 00 28 87 28 8f 00 70  │
 │  file sample.c                      │  │  002440 00 00 00 00 00 00 00 00 00 00 00  │
 │                                     │  │  003000 00 00 00 00 00 00 00 00 00 00 00  │
 │  >>>                                │  │                                        │
 └─────────────────────────────────────┘  └─────────────────────────────────────┘
```

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 '370.

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

4.2 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 10 different windows:

❑ 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).

❑ The **INSPECT window** displays trace samples and timing information.

❑ The **PROFILE window** displays statistics about code execution. This window is available only when you are in the profiling environment.

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

■ A MEMORY window displays the contents of a range of memory. You can display up to four MEMORY windows at one time.

■ A CPU window displays the contents of '370 CPU 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 4.4, *The Active Window*, on page 4-21.

The remainder of this section describes the individual windows.

COMMAND window
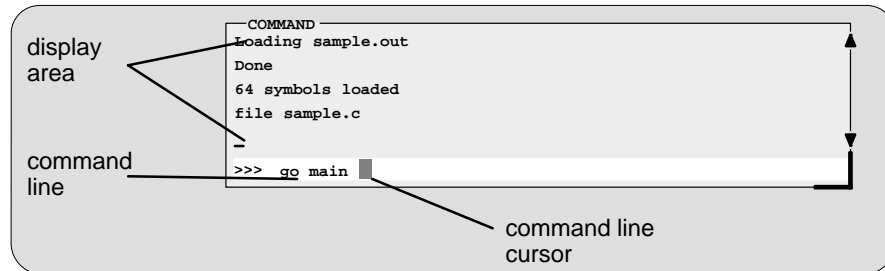
Purpose	❑ Provides an area for entering commands ❑ Provides an area for echoing commands and displaying command output, errors, and messages
Editable?	Command line is editable; command output isn't
Modes	All modes
Created	Automatically
Affected by	❑ All commands entered on the command line ❑ All commands that display 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 50 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.

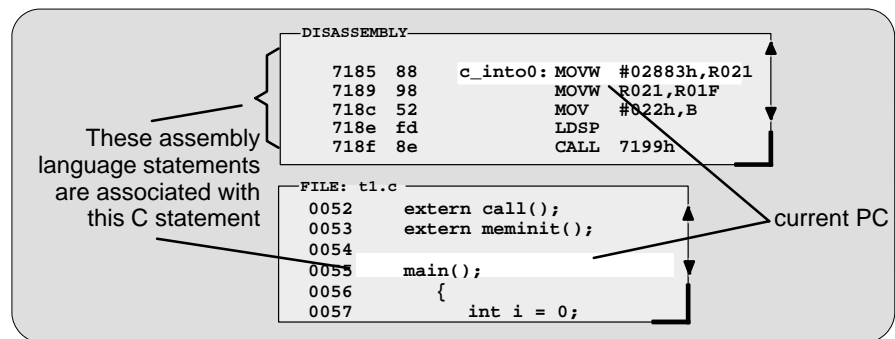For more information about the COMMAND window and entering commands, refer to Chapter 5.
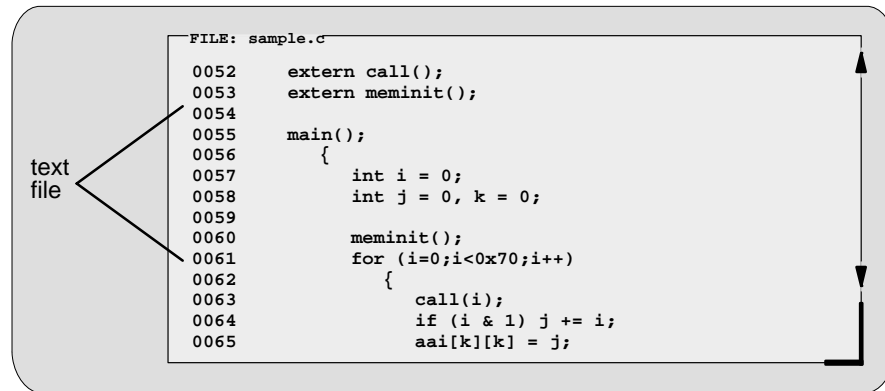
DISASSEMBLY window
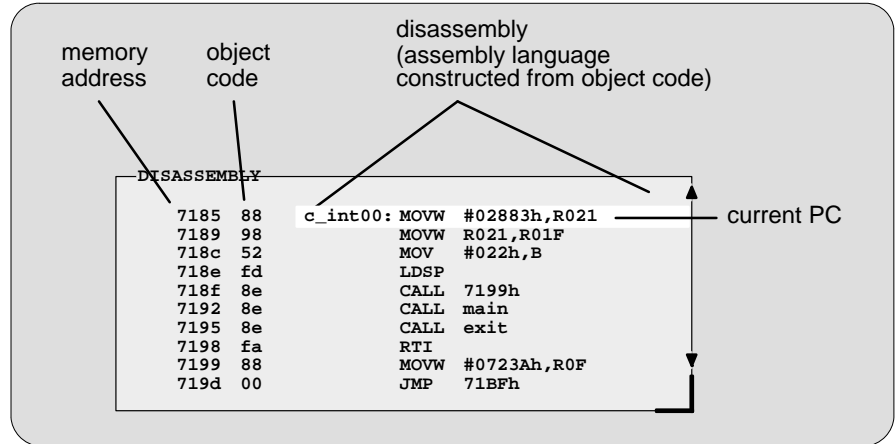
Purpose	Displays the disassembly (or reverse assembly) of memory contents
Editable?	No; pressing the edit key (⎡F9⎤) or the left mouse button sets a breakpoint on an assembly language statement
Modes	Auto (assembly display only), assembly, and mixed
Created	Automatically
Affected by	❏ DASM and ADDR commands
	❏ 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 with software breakpoints.

❏ The address and object code fields for all statements associated with the current C statement, as shown below

FILE window

```
┌─FILE: sample.c────────────────────────────────────┐  ▲
│ 0052    extern call();                             │
│ 0053    extern meminit();                          │
│ 0054                                               │
│ 0055    main();                                    │
│ 0056       {                                       │
│ 0057          int i = 0;                           │
│ 0058          int j = 0, k = 0;                    │
│ 0059                                               │
│ 0060          meminit();                           │
│ 0061          for (i=0;i<0x70;i++)                 │
│ 0062             {                                 │
│ 0063                call(i);                       │  ▼
│ 0064                if (i & 1) j += i;             │
│ 0065                aai[k][k] = j;                 │
└────────────────────────────────────────────────────┘
```

text file

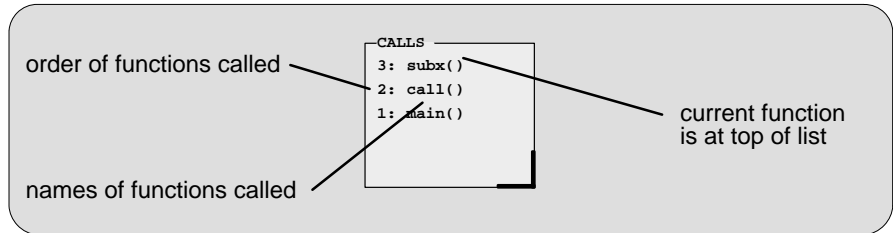|	
Purpose	Shows any text file you want to display
Editable?	No; Pressing the edit key (F9) or the left mouse button sets a breakpoint.
Modes	Auto (C display only) and mixed
Created	❑ With FILE command ❑ Automatically when you're in auto or mixed mode and your program begins executing C code
Affected 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

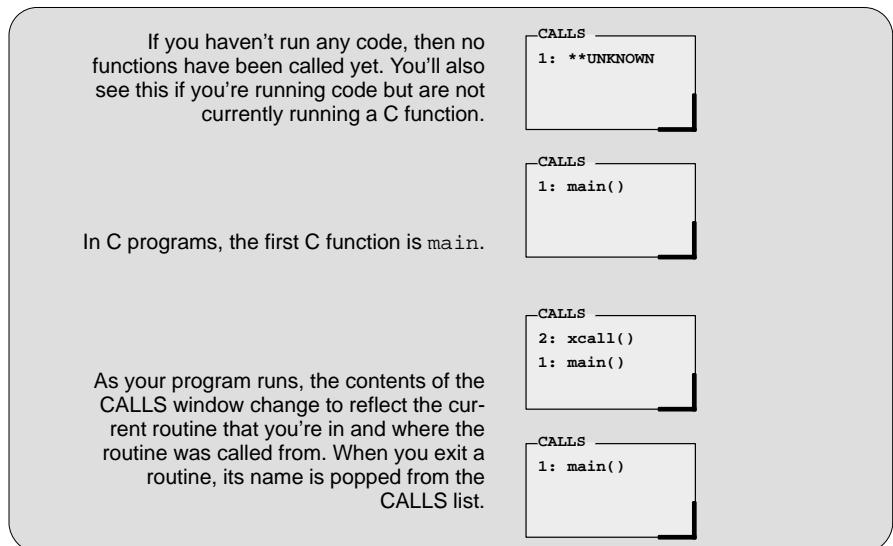## **CALLS window**

order of functions called

names of functions called

current function is at top of list

```
CALLS
3: subx()
2: call()
1: main()
```

Purpose	Lists the function you're in, its caller, and the caller's caller, etc., as long as each function is a C function
Editable?	No; you can't edit the window's contents
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 4.4, *The Active Window,* page 4-21).

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

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

INSPECT window

```
┌─ Inspect ─────────────────────────────────────────────────────────────────
│ INDX  ST   h   m  s ms  us  ns  EXTERNAL  CYCLE  ADDR  DATA  REVERSE ASM   ▲
│ 0000  0     0:00:00.000 000 400 11111111   IAQ   7003   8C   BR    7000h
│ 0001  0     0:00:00.000 000 800 11111111   READ  7004   70
│ 0002  0     0:00:00.000 001 200 11111111   READ  7005   00
│ 0003  0L    0:00:00.000 002 200 11111111   IAQ   7000   42   MOV   R07,R08
│ 0004  0     0:00:00.000 002 600 11111111   READ  7001   07
│ 0005  0     0:00:00.000 002 800 11111111   READ  0007   00                ▼
│ T1 0:00:00.000 000 000  AVG1 0:00:00.000 000 000  T2 0:00:00.000 000 000
└────────────────────────────────────────────────────────────────────────────
```

trace buffer

point & range
timer statistics

**XDS/22
emulator
only**

Purpose	Displays the contents of the trace buffer; also displays point and range timer statistics
Editable?	No
Modes	All
Created	By the INSPect command
Affected by	❏ B**TT**→**F**ormat menu selection
	❏ B**TT**→**P**osition menu selection
	❏ B**TT**→**L**ookup menu selection

The INSPECT window shows two types of information:

❏ The upper portion of the window displays the contents of the trace buffer. For a description of the fields in this area of the window, refer to Table 4–1.

❏ The lower portion of the window shows statistics for the point and range timers. Depending on which timer action you select first, one of the timers will be labeled as timer 1, and the other will be labeled as timer 2. The times listed in the INSPECT window are the total times for both timers, plus the average time for timer 1.

For more information about the INSPECT window, see Section 11.9, *Viewing Trace Buffer and Timing Information*, on page 11-20.
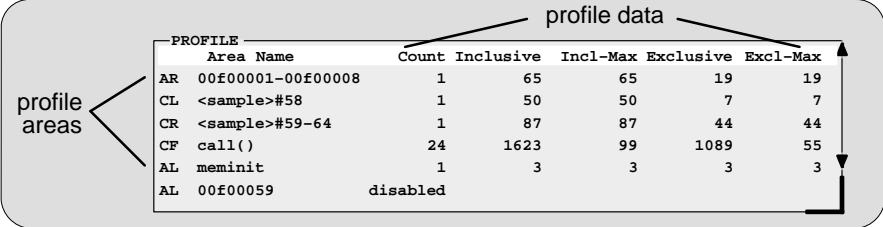
Table 4–1. Description of Trace Sample Information

Field	Description
INDX	Shows the trace sample's number within the trace buffer.
ST	Shows which state the BTT was in when it collected the trace sample. It will also show an E if the sample met BP/event conditions or an L if the sample was the last BP/event and caused a hardware breakpoint.
h–ns	Shows timing information about the trace sample. You can show one of three different types of timing: the total time since tracing began (default), the difference between current sample and previous sample, or the difference between current sample and any selected sample. To choose the type of time reporting, choose Format from the BTT menu.
EXTERNAL	Shows the values on the eight external probes.
CYCLE	Shows whether the cycle was a memory read (MR), memory write (MW), or instruction acquisition (IAQ).
ADDR	Shows the value on the address bus.
DATA	Shows the value on the data bus.
REVERSE ASM	Shows the associated assembly language code (if any).

XDS/22 emulator only

PROFILE window

profile data

```
┌─PROFILE
│     Area Name        Count Inclusive  Incl-Max Exclusive Excl-Max
│AR  00f00001-00f00008     1        65        65        19        19
│CL  <sample>#58           1        50        50         7         7
│CR  <sample>#59-64        1        87        87        44        44
│CF  call()               24      1623        99      1089        55
│AL  meminit               1         3         3         3         3
│AL  00f00059        disabled
```

profile areas

Purpose	Displays statistics collected during a profiling session
Editable?	No
Modes	Auto
Created	By invoking the debugger with the –profile option (you must be in a Microsoft Windows environment to use the profiler)
Affected by	❏ The PF and PQ commands ❏ Any commands on the View menu ❏ Clicking in the header area of the window

XDS/22 emulator only

The PROFILE window is visible only when you are in the profiling environment. The illustration above shows the window with a default set of data, but the display can be modified to show specific sets of data collected during a profiling session.

Note that within the profiling environment, the only other windows that are available are the COMMAND window, the DISASSEMBLY window, and the FILE window.

For more information about the PROFILE window (and about profiling in general), refer to Chapter 12, *Profiling Code Execution.*

MEMORY windows

```
    ┌─ MEMORY ──────────────────────────────────────────
    7185  88  28  83  21  98  21  1f  52  22  fd  8e  71
    7191  99  8e  70  00  8e  71  ce  fa  88  72  3a  0f
    719d  00  20  f4  ea  03  0f  d0  0d  f4  ea  02  0f
    71a9  d0  0c  70  04  0f  00  0a  9a  0f  9b  0d  70
    71b5  01  0d  70  01  0f  70  ff  0b  03  f1  f4  ea
    71c1  01  0f  d0  0b  9a  0f  d0  0a  14  0b  06  d2
    71cd  f9  8e  72  18  70  03  21  12  17  f4  eb  fe
    71d9  21  12  18  f4  eb  ff  21  12  19  9b  21  8a
```

addresses   data

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 (the default MEMORY window only)
	❏ You can display up to three additional MEMORY windows with the MEM# commands
Affected by	MEM commands: MEM, MEM1, MEM2, and MEM3.

A 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 MEMORY window above has twelve columns of data, so each new address is incremented by twelve. Although the window shows twelve columns of data, there is still only one column of addresses; the first value is at address 0x7185, the second at address 0x7191, etc.; the thirteenth value (first value in the second row) is at address 0x7215, etc.

As you run programs, some memory values change as the result of program execution. The debugger highlights the 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).

Three additional MEMORY windows called MEMORY1, MEMORY2, and MEMORY3 are available. The default MEMORY window does not have an extension number in its name; this is because MEMORY1, MEMORY2, and MEMORY3 are optional windows and can be opened and closed throughout your debugging session. Having four windows allows you to view four different memory ranges. Refer to Figure 4–4.

Figure 4–4. The Default and Additional MEMORY Windows
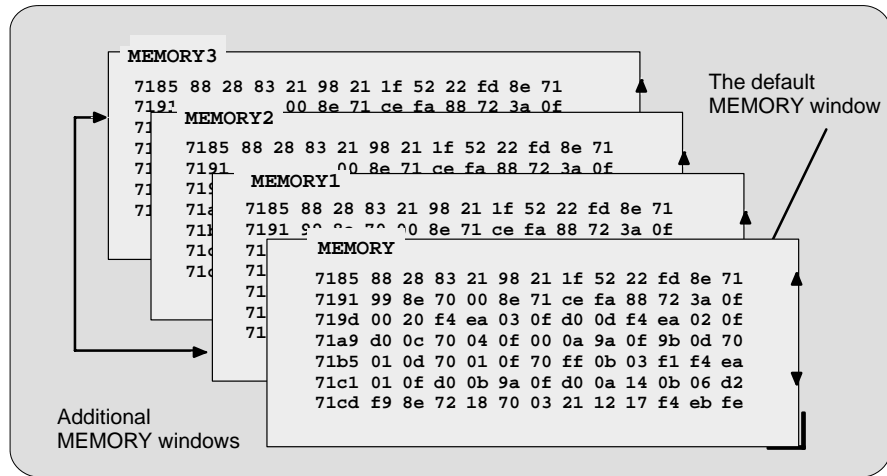
To create an additional MEMORY window or to display another range of memory in the current window, use the MEM command.

❑ **Creating a new MEMORY window.**

If the default MEMORY window is the only window open and you want to open another MEMORY window, enter the MEM command with the appropriate extension number:

mem[#] *address*

For example, if you want to create a new memory window starting at address 0x8000, you would enter:

mem1 0x8000 ⏎

This displays a new window, MEMORY1, showing the contents of memory starting at the address 0x8000.

❑ **Displaying a new memory range in the current MEMORY window.**

Displaying another block of memory identifies a new starting address for the memory range shown in the current MEMORY window. The debugger displays the contents of memory at *address* in the first data position in your MEMORY window. The end of the range is defined by the size of the window.

If the only memory window open is the default MEMORY window, you can view different memory locations by entering:

mem *address*

To view different memory locations in the optional MEMORY windows, use the MEM command with the appropriate extension number on the end. For example:

To do this. . .	Enter this. . .
View the block of memory starting at address 0x8000 in the MEMORY1 window	**mem1** 0x8000
View another block of memory starting at address 0x002f in the MEMORY2 window	**mem2** 0x002f

---

Note:

If you want to view a different block of memory explicitly in the default MEMORY window, you can use the alias command MEM0. This works *exactly* the same as the MEM command. To use this command, enter:

mem0 *address*

You can close and reopen additional MEMORY windows as often as you like.

❑ **Closing an additional MEMORY window.**

Closing a window is a two-step process:

1) Make the appropriate MEMORY window the active window (see Section 4.4, on page 4-21).

2) Press F4 .

Remember, you cannot close the default MEMORY window.

❑ **Reopening an additional MEMORY window.**

To reopen an additional MEMORY window after you've closed it, enter the MEM command with its appropriate extension number.

CPU window
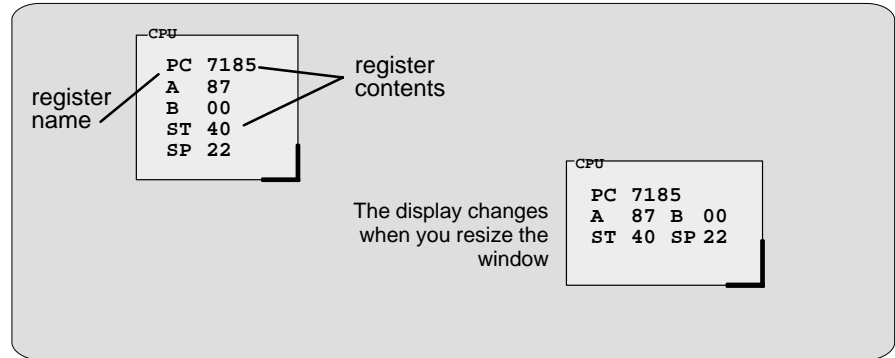
Purpose	Displays the contents of the '370 CPU registers
Editable?	Yes—you can edit the value of any displayed register
Modes	Auto (assembly display only), assembly, and mixed
Created	Automatically
Affected by	Data-management commands

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

DISP windows

```
           ┌─DISP: str────────────┐
           │ a    84              │
           │ b    86              │
structure  │ c    172            │        ┌─DISP: str.f4──────┐
members    │ f1   1              │        │ [0] 4427          │
           │ f2   7              │        │ [1] 1778          │
           │ f3   0x18           │        │ [2] 5554          │
member     │ f4   [...]          │        │ [3] 3567          │
values     └─────────────────────┘        │ [4] 1384          │
                                          │ [5] 1824          │
                                          │ [6] 3565          │
              This member is an array, and you   │ [7] 3774    │
              can display its contents in a sec-  │ [8] 1347   │
                        ond DISP window            │ [9] 1384  │
                                          └───────────────────┘
```
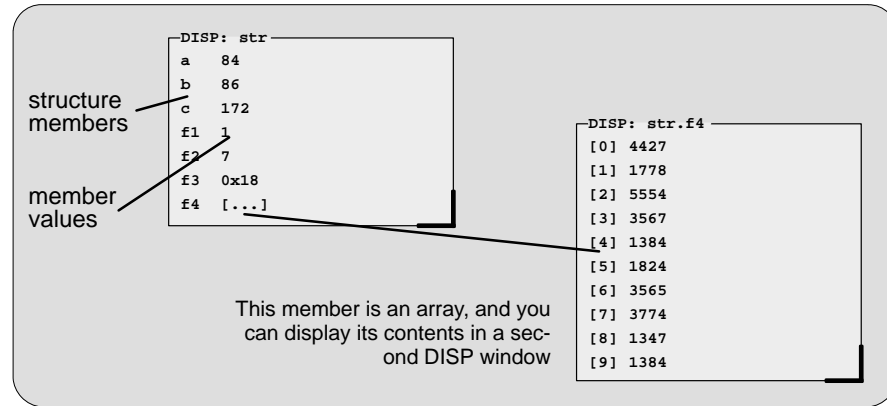
Purpose	Displays the members of a selected structure, array or pointer, and the value of each member
Editable?	Yes—you can edit individual values
Modes	Auto (C display only) and mixed
Created	With the DISP command
Affected by	The DISP command

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 in decimal.
❑ 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────────────┐
watch index────────────────│1:  R0    0x02    │ ▲
                           │2:  X+X   4       │ ▼
                           │3:  PC    0x4044  │
                           └──────────────────┘ ▐
                              │         │
                              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*] [,*display format*]]

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.

4.3 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──────────────────────────────────────────┐
│ '370 Debugger Version 4.01                      ▲│
│ Copyright (c) 1990, 1992 Texas Instruments       │
│ TMS370 Silicon Revision B                        │
│ Loading sample.out                               │
│   67 symbols loaded                             ▼│
│ Done                                             │
│ _file sample.c                                   │
│ >>> go main ▮                                    │
└──────────────────────────────────────────────────┘
                              command line cursor
```
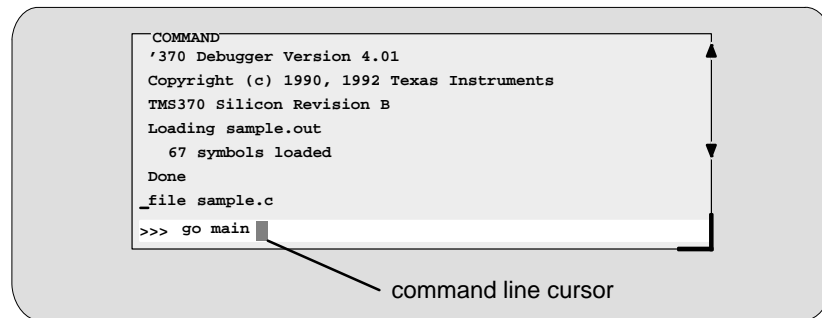
❏ The **mouse cursor** is a block-shaped cursor that tracks mouse movements over the entire display. This cursor is controlled by the mouse driver installed on your system; if you haven't installed a mouse, you won't see a mouse cursor on the debugger display.

❏ 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──────────┐
│   PC  7185  ▲│
│   A   87     │
│   B   00    ▼│
│   ST  40     │
│   SP  22     │
└──────────────┘
                current field cursor
```
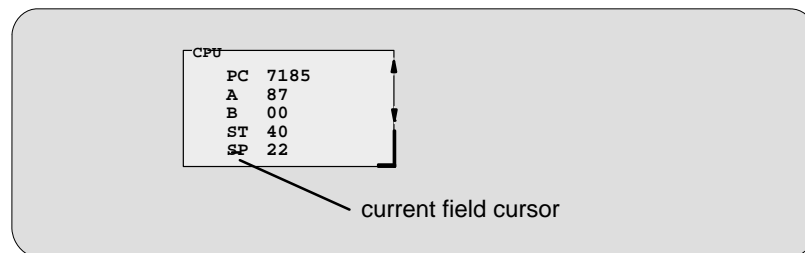
4.4 The Active Window

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 4–5 illustrates the default appearance of an active window and an inactive window.

Figure 4–5. Default Appearance of an Active and an Inactive Window

An active window (default appearance)

```
══COMMAND══
'370 Debugger Version 4.01
Copyright (c) 1990, 1992 Texas Ins
TMS370 Silicon Revision B
Loading sample.out
  67 symbols loaded
>>>
```

This window is highlighted to show that it is active

An inactive window (default appearance)

```
──COMMAND──
TMS370 Silicon Revision B
'370 XDS v2.06 PT v1.4
Loading sample.out
Done
go main
>>>
```

This window is not highlighted and is not active

Note: On **black-and-white 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 when the window becomes active).

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 and click within the window, you might also select the cur-
         rent field. For example,

         ❏ If you point and click inside the CPU window, then the register you're point-
             ing 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 and click inside the DISASSEMBLY or FILE window, you'll set
             a breakpoint on the statement you're pointing to.

             Press the mouse button again to clear the breakpoint.

F6   This key cycles 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 nec-
         essary 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 to 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.

4.5 Manipulating Windows

A window's size and its position in the debugger display aren't fixed—you can resize and move windows.

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 4.4 (page 4-21).
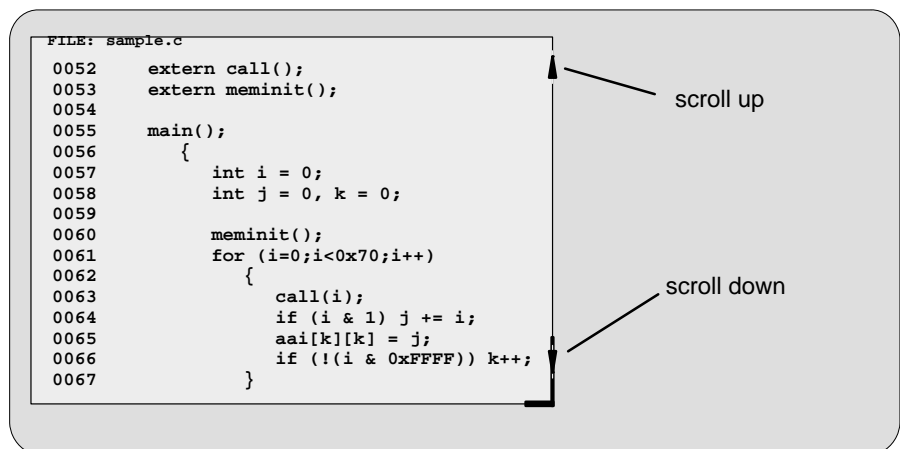

Resizing a window

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────────────┐
│ '370 Debugger Version 1.00
│ Copyright (c) 1990, 1992 Texas
│ TMS370 Silicon Revision B
│ Loading sample.out
│   67 symbols loaded
│ Done
│ file sample.c
│>>>
└───────────────────┘
```
lower right corner (highlighted)
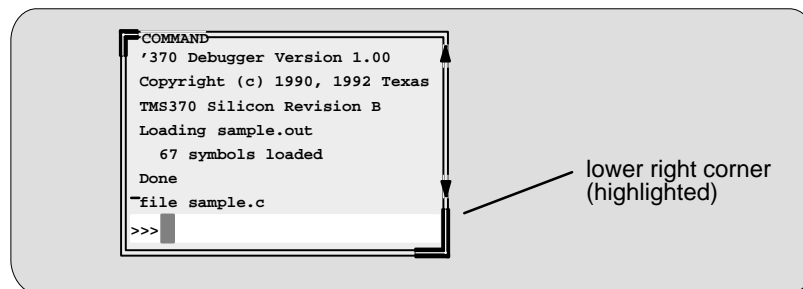
2) Grab the highlighted corner by pressing one of the mouse buttons; while pressing the button, move the mouse in any direction. This resizes the window.

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 the *width* and *length* parameters. Valid values for the width and length depend on the screen size and the window position on the screen. If the window is in the upper left corner of the screen, the maximum size of the window is the same as the screen size minus one line. (The extra line is needed for the menu bar.) For example, if the screen size is 80 characters by 25 lines, the largest window size is 80 characters by 24 lines.

If a window is in the middle of the display, 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 4-26.

For example, 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 or unzoom a window:

❏ By using the mouse.

❏ 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:

upper left corner (highlighted)

```
COMMAND
'370 Debugger Version 1.00
Copyright (c) 1990, 1992 Texas
TMS370 Silicon Revision B
Loading sample.out
_ 67 symbols loaded
>>>
```
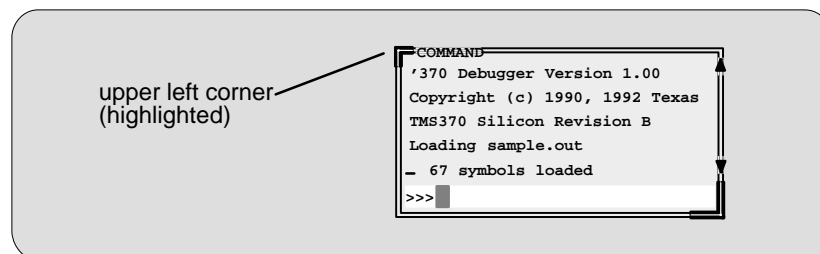
2) Click the left mouse button.

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.

Point to the top edge or the left edge

```
 COMMAND
Copyright (c) 1990, 1992 Texas
TMS370 Silicon Revision B
Loading sample.out
  67 symbols loaded
Done
>>>
```
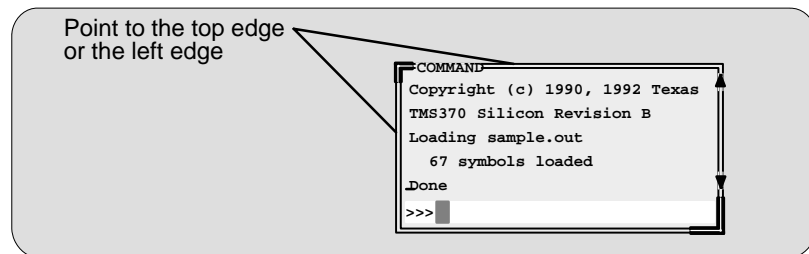
⊐▯ 2) Press the left mouse button, but don't release it; now move the mouse in any direction.

▯▯ 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. X positions are valid if the X position plus the window width in characters is less than or equal to the screen width in characters. Y positions are valid if the Y position plus the widow height is less than or equal to the screen height in lines.

For example, if the window is 10 characters wide and 5 lines high and the screen size is 80 x 25, the command **move 70, 20** would put the lower right-hand corner of the window in the lower right-hand corner of the screen. No X value greater than 70 or Y value greater than 20 would be valid in this example.

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.

4.6 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 4.4 (page 4-21).

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
0052     extern call();
0053     extern meminit();
0054
0055     main();
0056        {
0057           int i = 0;
0058           int j = 0, k = 0;
0059
0060           meminit();
0061           for (i=0;i<0x70;i++)
0062              {
0063                 call(i);
0064                 if (i & 1) j += i;
0065                 aai[k][k] = j;
0066                 if (!(i & 0xFFFF)) k++;
0067              }
```

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 mov-
       ing 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 struc-
       tures 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 con-
         tents 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 8.3, page 8-4.)

Note:

In the FILE, DISASSEMBLY, CALLS, and PROFILE windows, the "click and type" method of selecting data for editing—pointing at a line and pressing F9 or the left mouse button—does not allow you to modify data.

❑ In the FILE and DISASSEMBLY windows, pressing F9 or the 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 the mouse button shows the source for the function named on the selected line.

❑ In the PROFILE window, pressing F9 has no effect. Clicking the mouse button in the header displays a different set of data; clicking the mouse button on an area name shows the code associated with the area.

4.7   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, WATCH, and MEMORY windows.

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

❏ To close the CALLS or INSPECT window:

1)   Make the CALLS or INSPECT 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 an additional MEMORY window:

1)   Make the appropriate MEMORY window the active window.

2)   Press ⌷F4⌷ .

Note:

You cannot close the default MEMORY window.

❏ To close the WATCH window, enter:

wr ⌷⏎⌷

When you close a window, the debugger remembers the window's size and position. The next time you open the window, it will have the same size and position. That is, if you close the CALLS window, then reopen it, it will have the same size and position as it did before you closed it. Since you can open numerous DISP and MEM windows, when you open one, it will occupy the same position as the last one of that type that you closed.

Entering and Using Commands

The debugger provides you with several methods for entering commands:

❑ From the command line
❑ From the pulldown menus (using keyboard combinations or the mouse)
❑ With function keys
❑ From a batch file

Mouse use and function key use differ from situation to situation; their use is described throughout this book whenever applicable. This chapter includes specific rules that apply to entering and using pulldown menus. Also included is information about entering DOS commands and defining your own command strings.

5.1 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 various sections throughout this book, as they apply to the current topic. Chapter 13 summarizes all of the debugger commands with an alphabetical 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 5–1 shows the COMMAND window.

Figure 5–1. The COMMAND Window
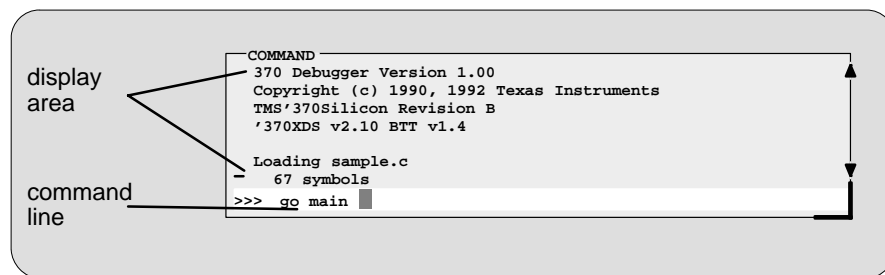
The COMMAND window serves two purposes.

❑ The **command line** portion of the window provides you with an area for entering commands. For example, the command line in Figure 5–1 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 5–1 shows the messages that are displayed when you first bring up the debugger and also shows that a file was loaded.

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 at the current field in the box. Refer to Section 5.3, on page 5-11, for more information on dialog boxes.

Using the command history

The debugger keeps an internal list, or **command history**, of the commands that you enter. It remembers the last 50 commands that you entered. If you want to reenter 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 on the command line, 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

Recording information from the display area

The information shown in the display area of the COMMAND window can be written to a log file. The log file is a text file that contains commands you've entered, their results, and error or progress messages. To record this information in a log file, use the DLOG command.

Log files can be executed by using the TAKE command. When you use DLOG to record the information from the COMMAND window display area, the debugger automatically precedes all error or progress messages and command results with a semicolon to turn them into comments. This way, you can easily re-execute the commands in your log file by using the TAKE command.

❑ To begin recording the information shown in the COMMAND window display area, use:

dlog *filename*

This command opens a log file called *filename* that the information is recorded into.

❑ To end the recording session, enter:

dlog close ⏎

If necessary, you can write over existing log files or append additional information to existing files. The extended format for the DLOG command is:

dlog *filename* [**,**{**a** | **w**}]

The optional parameters of the DLOG command control how the log file is created and/or used:

❑ **Creating a new log file.** If you use the DLOG command without one of the optional parameters, the debugger creates a new file that it records the information into. If you are recording to a log file already, entering a new DLOG command and filename closes the previous log file and opens a new one.

❑ **Appending to an existing file.** Use the **a** parameter to open an existing file to which to append the information in the display area.

❑ **Writing over an existing file.** Use the **w** parameter to open an existing file to write over the current contents of the file. Note that this is the default action if you specify an existing filename without using either the **a** or **w** options; you will lose the contents of an existing file if you don't use the append (a) option.

5.2   Using the Menu Bar and the Pulldown Menus

In all three of the debugger modes, 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 5–2 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 5–2. The Menu Bar in the Basic Debugger Display

Figure 5–3. All of the Pulldown Menus (Basic Debugger Display)
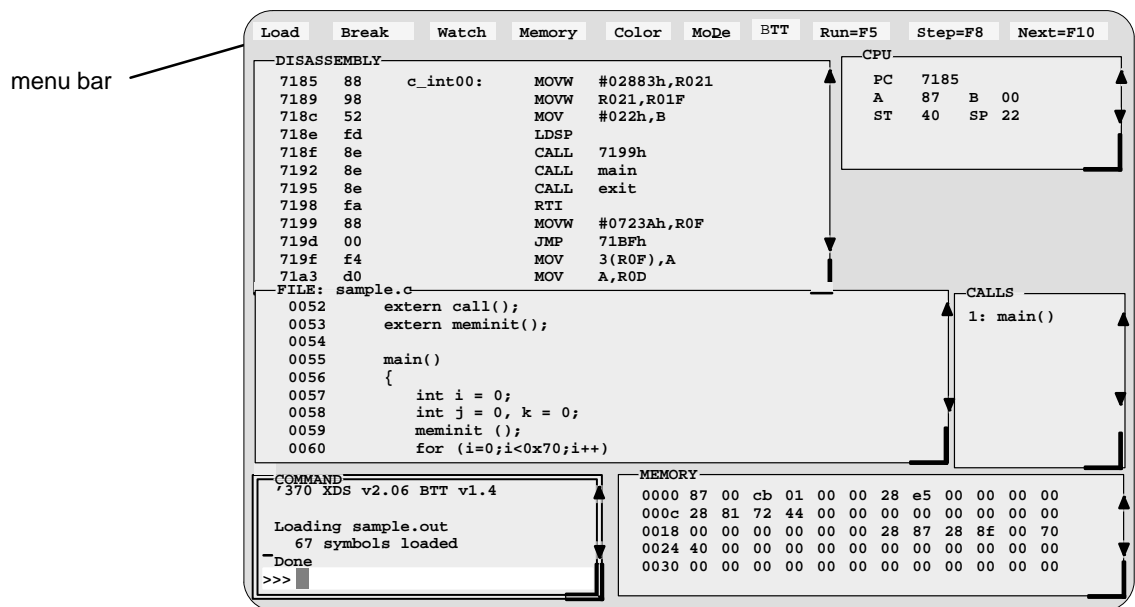
Several of the selections on the menu bar have pulldown menus; if they could all be pulled down at once, they'd look like Figure 5–3.
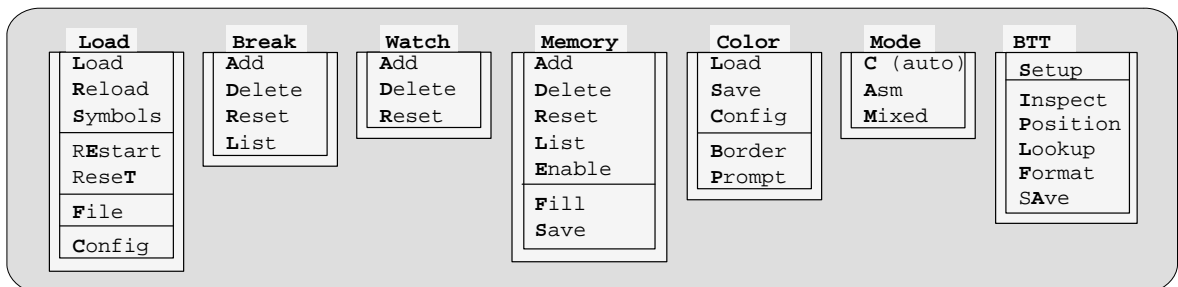
Note: The BTT menu is available only when you are using the XDS/22 emulation system.

Pulldown menus in the profiling environment

The debugger displays a different menu bar in the profiling environment:

```
Load    mAp   Mark   Enable   Disable   Unmark   View   Stop-points   Profile
```

The Load menu corresponds to the Load menu available in the basic debugger environment. The other entries provide access to profiling commands. The mAp menu provides memory map commands available from the basic Memory menu.

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.

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)   Press and release 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.

5.3 Using Dialog Boxes

Many of the debugger commands have parameters. When you execute these commands from pulldown menus, you must have some way of providing parameter information. The debugger allows you to do this by displaying a **dialog box** that asks for this information.

Some debugger commands have very simple dialog boxes that provide you with an alternative method for typing in values. Other commands, such as BTT commands, have more complex dialog boxes; in addition to typing in values, you may be asked to make selections from a list of predefined parameters.

Entering text in a dialog box

Entering text in a dialog box is much like entering commands on the command line. For example, the Add entry on the Watch menu is equivalent to entering the WA command. This command has three parameters:

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

When you select Add from the Watch 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
                                    <<OK>>   <CANCEL>
└─────────────────────────────────────────────────────┘
```

You can enter an *expression* just as you would if you were to type the WA command, and then press ⌷TAB⌷ or ↓. The cursor moves down to the next parameter:

```
┌ Watch Add ─────────────────────────────────────────┐
  Expression  MY_VAR
  Label
  Format
                                    <<OK>>   <CANCEL>
└─────────────────────────────────────────────────────┘
```

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.

In the case of the WA command, the 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 optional parameters, don't type anything in their fields—just continue to the next parameter.

Modifying text in a dialog box is similar to editing text on the command line:

❏ 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 `TAB` or ⊥ to move to the next parameter.

❏ You can edit what you type (or values that remain from a previous entry) in the same way that you can edit text on the command line. See Section 5.1 for more information on editing text on the command line.

When you've entered a value for the final parameter, point and click on <OK> to save your changes, or <CANCEL> to discard your changes; the debugger closes the dialog box and executes the command with the parameter values you supplied.

Selecting parameters in a dialog box

More complex dialog boxes, such as those associated with BTT commands, allow you to:

❏ **Enter text.** Entering text in a more complex dialog box is the same as entering text on the command line. Refer to the discussion above, *Entering text in a dialog box*, for more information.

❏ **Choose from a list of predefined options.** There are two types of predefined options in a dialog box. The first type of option allows you to enable one or more predefined options. The second type of option is *mutually exclusive*; therefore, you can enable only one at a time.

Valid options (of the opened dialog box) are listed for you so that all you have to do is point and click to make your selections.

❏ **Close the dialog box.** The more complex dialog boxes do not close automatically. They allow you the option of saving or discarding any changes you made to your parameter choices. All you have to do to close the dialog box is point and click on the appropriate option, either OK, CANCEL, or DELETE.

Figure 5–4 shows you the components of a complex dialog box used with the BTT.

Figure 5–4. The Components of a Dialog Box
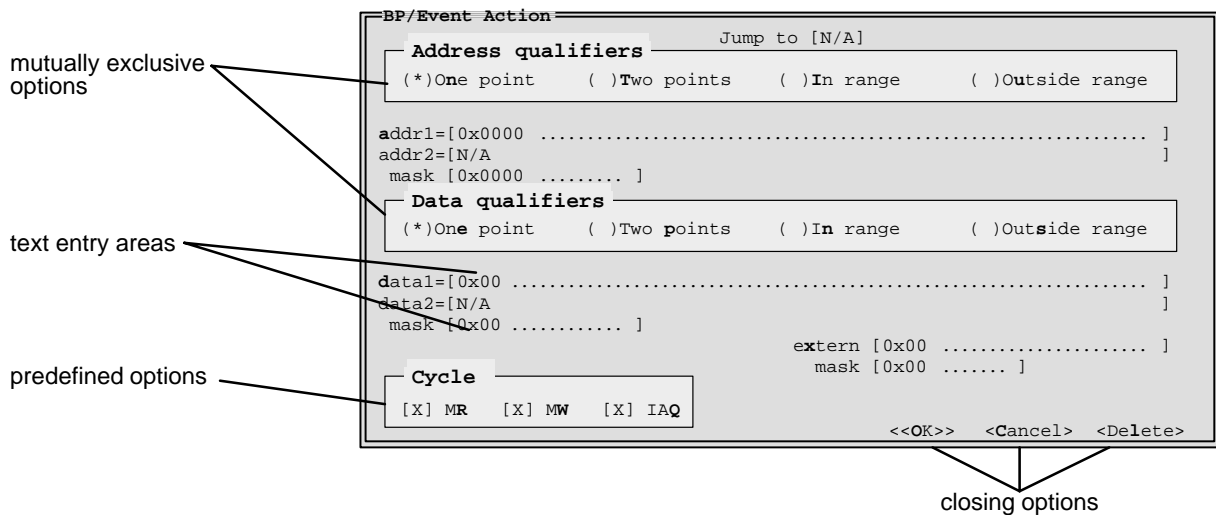
When you display a dialog box for the first time during a debugging session, nothing is enabled. When you bring up the same dialog box again, though, your previous selections are remembered. (This is similar to having a command history.)

As Figure 5–4 shows, options are preceded by either square brackets or parentheses; mutually exclusive options are only preceded by parentheses. Enabling options preceded by square brackets is like turning a switch on and off. When the option is enabled, the debugger displays an X inside the brackets preceding the option. You can enable as many of these options as you want:

```
[X]   Option 1    [ ] Option 2   [X] Option 3
[ ]   Option 4    [X] Option 5   [X] Option 6
[X]   Option 7    [ ] Option 8   [ ] Option 9
```
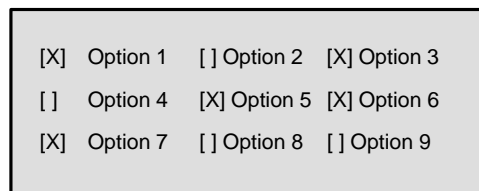
Mutually exclusive options, however, are enabled when the debugger displays an asterisk inside the parentheses preceding your selection. The following example illustrates this:

> (*)   Option 1
>
> ( )   Option 2
>
> ( )   Option 3
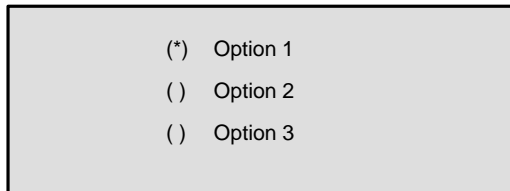
Notice that only one option is enabled at a time. There are several ways to enable both types of options:

1)   Point the cursor at the option you want to enable.

2)   Click the left mouse button. This enables the event and displays an X next to the option (or an asterisk next to a mutually exclusive option).

Repeat these two steps to disable an option. When the X (or asterisk) is no longer displayed, that option has been disabled.

Keyboard Method 1

(ALT)   1)   Press the (ALT) key; don't release it.

(X)   2)   Press and release the key that corresponds to the highlighted letter or number of the option you want to enable. The debugger displays an X (or asterisk) next to the option, indicating that selection is enabled.

Repeat these two steps to disable an option. When the X (or asterisk) is no longer displayed, that option has been disabled.

Keyboard Method 2

TAB   1)   Press the TAB key to move throughout the dialog box until your cursor points to the option you want to enable.

↓ ↑   2)   Use the arrow keys to move up and down or left and right.

When you enable a mutually exclusive option, moving the arrow keys alone will place an asterisk inside the parentheses, indicating that the option is enabled. However, to enable an option preceded by square brackets, you must:

SPACE   Press the SPACE bar. The debugger displays an X next to your selection, thus enabling that particular option.

or

F9   Press the F9 key. The debugger displays an X next to your selection, thus enabling that particular option.

Repeat these steps to disable a option.

Closing a dialog box

The more complex dialog boxes do not close automatically; the debugger expects input from you. When you close a dialog box, you can:

❑   Save the changes you made

or   ❑   Discard any of the changes you made

Note:

The default option, <<OK>>, is highlighted; clicking on this option saves your changes and closes the dialog box.

There are several ways to close a dialog box:

↖   1)   Point the cursor at <<OK>> to close the dialog box and save your changes. Or you can opt to discard your changes by pointing the cursor at <<CANCEL>>.

2)   Click the left mouse button. This executes your choice and closes the dialog box.

Keyboard Method 1

(ALT)   1)   Press the (ALT) key; don't release it.

(X)   2)   Press and release the (O) key to save your changes. Press and release the (A) key to discard your changes. Both of these actions execute your choice and close the dialog box.

Keyboard Method 2

(TAB)   1)   Press the (TAB) key to move through the dialog box until your cursor is in the <<OK>> or <<CANCEL>> field.

(←) (→)   2)   Use the arrow keys to switch between <<OK>> and <<CANCEL>>.

(↵)   3)   Press the (↵) key to accept your selection. This executes your choice and closes the dialog box.

5.4 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 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=*pathname; pathname*

This allows you to name several directories that the debugger can search. If you often use the same directories, it may be convenient to set D_DIR in your autoexec.bat file. You can also set D_DIR from within the debugger by using the SYSTEM command (see Section 5.6, *Entering Operating-System Commands*, page 5-23).

❑ 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.

Echoing strings in a batch file

When executing a batch file, you can display a string to the COMMAND window by using the ECHO command. The syntax for the command is:

echo *string*

This displays the *string* in the display area of the COMMAND window.

For example, you may want to document what is happening during the execution of a certain batch file. To do this, you could use the following line in your batch file to indicate that you are creating a new memory map for your device:

```
echo Creating new memory map
```

(Notice that the string should not be in quotes.)

When you execute the batch file, the following message appears:

```
.
.
Creating new memory map
.
.
```

Note that any leading blanks in your string are removed when the ECHO command is executed.

Controlling command execution in a batch file

In batch files, you can control the flow of debugger commands. You can choose to conditionally execute debugger commands or set up a looping situation by using IF/ELSE/ENDIF or LOOP/ENDLOOP, respectively.

❏ To conditionally execute debugger commands in a batch file, use the IF/ELSE/ENDIF commands. The syntax is:

if *Boolean expression*
debugger command
debugger command

.

.
[**else**
debugger command
debugger command

.

.]
endif

The debugger includes some predefined constants for use with IF. These constants evaluate to 0 (false) or 1 (true). Table 5–1 shows the constants and their corresponding tools.

Table 5–1. Predefined Constants for Use With Conditional Commands

Constant	Debugger Tool
$$XDS22$$	emulator
$$ABD$$	application board

If the Boolean expression evaluates to true (1), the debugger executes all commands between the IF and ELSE or ENDIF. Note that the ELSE portion of the command is optional. (See Chapter 14 for more information about expressions and expression analysis.)

One way you can use these predefined constants is to create an initialization batch file that works for any debugger tool. This is useful if you are using, for example, both the emulator and the application board. To do this, you can set up the following batch file:

```
if $$XDS22$$
echo Invoking initialization batch file for emulator.
use \xds370
take init.cmd
.
.
.
endif

if $$ABD$$
echo Invoking batch file for application board.
use \abd370
take init.cmd
.
.
.
endif
.
.
.
```

In this example, the debugger will execute only the initialization commands that apply to the debugger tool that you invoke.

❑ To set up a looping situation to execute debugger commands in a batch file, use the LOOP/ENDLOOP commands. The syntax is:

loop *expression*
debugger command
debugger command

.

.

endloop

These looping commands evaluate in the same method as in the run conditional command expression. (See Chapter 14 for more information about expressions and expression analysis.)

■ If you use an *expression* that is not Boolean, the debugger evaluates the expression as a loop count. For example, if you wanted to execute a sequence of debugger commands ten times, you would use the following:

```
loop 10
runb
.
.
.
endloop
```

The debugger treats the 10 as a counter and executes the debugger commands ten times.

■ If you use a Boolean *expression*, the debugger executes the commands repeatedly as long as the expression is true. This type of expression has one of the following operators as the highest precedence operator in the expression:

>	>=	<
<=	==	!=
&&	||	!

For example, if you want to continuously trace some register values, you can set up a looping expression like the following:

```
loop !0
step
? PC
? SP
endloop
```

The IF/ELSE/ENDIF and LOOP/ENDLOOP commands work with the following conditions:

❑ You can use conditional and looping commands only in a batch file.

❑ You must enter each debugger command on a separate line in the batch file.

❑ You can't nest conditional and looping commands within the same batch file.

5.5 Defining Your Own Command Strings

The debugger provides a shorthand method of entering often-used commands or command sequences. This process 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 0x014,0x18,0x11
```

The first value (0x014) 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.

❑ **Redefining an alias.** To redefine an alias, re-enter the ALIAS command with the same alias name and a new command string.

❑ **Deleting aliases.** To get rid of a single alias, use the UNALIAS command:

unalias *alias name*

To delete *all* aliases, enter the UNALIAS command with an asterisk instead of an alias name:

unalias *

Note that the * symbol *does not* work as a wildcard.

Note:

❑ Alias definitions are lost when you exit the debugger. If you want to reuse aliases, define them in a batch file.

❑ 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.

5.6 Entering Operating-System Commands

The debugger provides a simple method of entering DOS commands without explicitly exiting the debugger environment. To do this, use the SYSTEM command. The format for this command is:

system   [*DOS 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 SYSTEM command with a DOS command as a parameter, then you stay within the debugger environment.

❏ If you enter the SYSTEM 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 DOS command, supply it as a parameter to the SYSTEM command. For example, 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 DOS 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 DOS 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 opens a system shell to display the following message:

```
         1 File(s) copied
Type Carriage Return To Return To Debugger
```

The message displays until you press ⏎.

If you want the debugger to display the message and then return immediately to the debugger environment, you can 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 DOS prompt. At this point, you can enter any DOS command.

When you are finished entering commands and are ready to return to the debugger environment, enter:

exit ⏎

Note:

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 be entered using the Memory pulldown menu.

6.1 The Memory Map: What It Is and Why You Must 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. Typically, the map matches the MEMORY definition in your linker command file.

A special default initialization batch file included with the debugger package defines a memory map for your version of the debugger. This memory map may be sufficient when you first begin using the debugger. However, the debugger provides a complete set of memory-mapping commands that let you modify the default memory map or define a new memory map.

You can define the memory map interactively by entering the memory-mapping 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 defining a memory map is to put the memory-mapping commands in a batch file.

Defining the memory map in a batch file

There are two methods for defining the memory map in a batch file:

❏ You can redefine the memory map in the initialization batch file.

❏ You can define a memory map in a separate batch file of your own.

When you invoke the debugger, it follows these steps to find the batch file that defines your memory map:

1) When you invoke the debugger, it checks to see if you've used the –t debugger option. The –t option allows you to specify a batch file other than the initialization batch file shipped with the debugger. If it finds the –t option, the debugger reads and executes the specified file.

2) If you don't use the –t option, the debugger looks for a file called *init.cmd*. If the debugger finds this file, it reads and executes the commands.

Potential memory map problems

The following are potential problems you may experience if the memory map isn't correctly defined and enabled:

❑ **Accessing invalid memory addresses.** If you don't supply a batch file containing memory-map commands, then the debugger is initially unable to access any target memory locations. 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.)

❑ **Accessing an undefined or protected area.** 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.

6.2   A Sample Memory Map

Because you must define a memory map before you can run any programs, it's convenient to define the memory map in the initialization batch file. Figure 6–1 *(a)* shows a sample of the memory map commands defined in an initialization batch file. If you are using the XDS/22, you can use the file as is, edit it, or create your own memory map batch file. The files shipped with the application board are similar to that of the XDS/22.

The MA commands define valid memory ranges and identify the read/write characteristics of the memory ranges. The MAP command enables mapping (note that by default, mapping is enabled when you invoke the debugger). Figure 6–1 *(b)* illustrates the memory map defined by the default batch file.

Figure 6–1. Sample Memory Map for Use With an Emulator

(a) Memory map commands

```
ma      0, 0x100, iram
ma      0x100, 0x100, eram
ma      0x1010, 0x10, iram
ma      0x1020, 0x10, iram
ma      0x1030, 0x10, siper
ma      0x1040, 0x10, tiper
ma      0x1050, 0x10, siper
ma      0x1060, 0x10, tiper
ma      0x1070, 0x10, iram
ma      0x2000, 0x1000, eram
ma      0x4000, 0x4000, erom
```

(b) Memory map defined by a sample batch file

Address	Description
0000 – 00FFh	internal RAM
0100 – 01FFh	emulator RAM
1010 – 101Fh	internal RAM
1020 – 102Fh	internal RAM
1030 – 103Fh	serial internal peripheral frame
1040 – 104Fh	timer internal peripheral frame
1050 – 105Fh	serial internal peripheral frame
1060 – 106FH	timer internal peripheral frame
1070 – 107Fh	internal RAM
2000 – 3FFFh	emulator RAM
4000 – 7FFFh	emulator ROM

6.3 Identifying Usable Memory Ranges

ma The debugger's MA command identifies valid ranges of target memory. The syntax of the MA command is:

ma *address, length, type*

❏ The *address* parameter defines the starting address of a range. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.

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
```

❏ 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. You can identify three basic kinds of memory:

■ **Internal memory** accesses memory locations that are internal to the '370 device inside the emulator.

■ **External memory** accesses memory locations on the target system. An example of this might be a target system with memory expansion capabilities, in which the expanded memory resides on the target system and is external to the emulator.

■ **Emulator memory** accesses memory locations only inside the emulator, not inside the '370 device. For example, if you are using a ROM-less device without a target system, you can emulate the external memory inside the emulator.

To identify this kind of memory, *type* must be one of these keywords:

To identify this kind of memory,	Use this keyword as the *type* parameter
read-only emulator memory	**R, ROM, EROM**
read-only external memory	**XROM**
read-only internal memory	**IROM**
read/write emulator memory	**RW, RAM, ERAM**
read/write external memory	**XRAM**

To identify this kind of memory,	Use this keyword as the *type* parameter
read/write internal memory	**IRAM**
read/write serial peripheral frame in emulator memory	**SERW, SEPER**
read/write serial peripheral frame in internal memory	**SIRW, SIPER**
read/write timer peripheral frame in emulator memory	**TERW, TEPER**
read/write timer peripheral frame in internal memory	**TIRW, TIPER**
inaccessible memory	**PROTECT**
EPROM control frame	**EPCTL**
program EPROM read-only emulator memory	**PEPROM**
data EPROM read-only emulator memory	**DEPROM**
custom EPROM read-only emulator memory	**CEPROM**
program EEPROM read-only emulator memory	**PEEPROM**
data EEPROM read-only emulator memory	**DEEPROM**
custom EEPROM read-only emulator memory	**CEEPROM**

Restrictions on usable memory ranges

The following restrictions apply to identifying usable memory ranges:

❏ You should always start the map for a peripheral frame or an EEPROM control frame on an address ending in 0 and give it a length that is a multiple of 16.

❏ You must define an EPROM control frame in order to define a type EEPROM/EPROM.

❏ You can define only one EPROM control frame (it *must* have a length of 0x10).

❏ You can define only one data EEPROM or EPROM, one program EEPROM or EPROM, or one custom EEPROM or EPROM (for a maximum of three types). For example, if you have already defined a type DEPROM, you cannot define a type DEEPROM.

6.4 Enabling Memory Mapping

map By default, mapping is enabled when you invoke the debugger. In some instances, you may want to explicitly enable or disable memory. You can use the MAP command to do this; the syntax for this command is:

> **map on**
>
> or **map off**

Disabling memory mapping can cause bus fault problems in the target system because the debugger may attempt to access nonexistent memory.

Note:

When memory mapping is enabled, you cannot access memory locations that are not defined by an MA command.

If you attempt to access memory in these situations, the debugger displays this message in the COMMAND window display area:

```
Error in expression
```

6.5 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, the ending address, the read/write characteristics, and the type of memory for each defined memory range. For example, if you're using the sample memory map for the emulator (shown in Figure 6–1 *(b)*) and you enter the ML command, the debugger displays:

```
Memory range              Attributes
0000 – 00ff               INT         READ  WRITE
0100 – 01ff               EMU         READ  WRITE
1010 – 101f               INT         READ  WRITE
1020 – 102f               INT         READ  WRITE
1030 – 103f               SERIAL INT  READ  WRITE
1040 – 104f               TIMER INT   READ  WRITE
1050 – 105f               SERIAL INT  READ  WRITE
1060 – 106f               TIMER INT   READ  WRITE
1070 – 107f               INT         READ  WRITE
2000 – Efff               EMU         READ  WRITE
4000 – 7fff               EMU         READ
```

6.6 **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
```

> **Note:**
>
> When you remove an EEPROM/EPROM control frame, any defined EEPROMs/EPROMs are also removed.

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 6-5.

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.

6.7 Using Multiple Memory Maps for Multiple Target 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 initialization batch 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 target 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 initialization batch file as usual. 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.

Chapter 7

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.

7.1 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's selection of the appropriate display is 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 4.1, *Debugging Modes and Default Displays* (page 4-2).

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
```
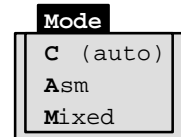
1) Point to the menu name.

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 5.2, *Using the Pulldown Menus,* on page 5-7.

F3   Pressing this key causes the debugger to switch modes in this order:

$$\text{auto} \longrightarrow \text{assembly} \longrightarrow \text{mixed}$$
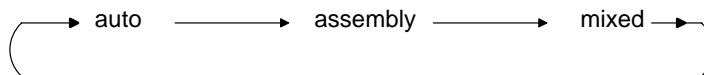
Enter any of these commands to switch to the desired debugging mode:

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.

7.2 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 memory contents. (This code doesn't come from any of your text files or from the intermediate assembly files produced by the compiler.)
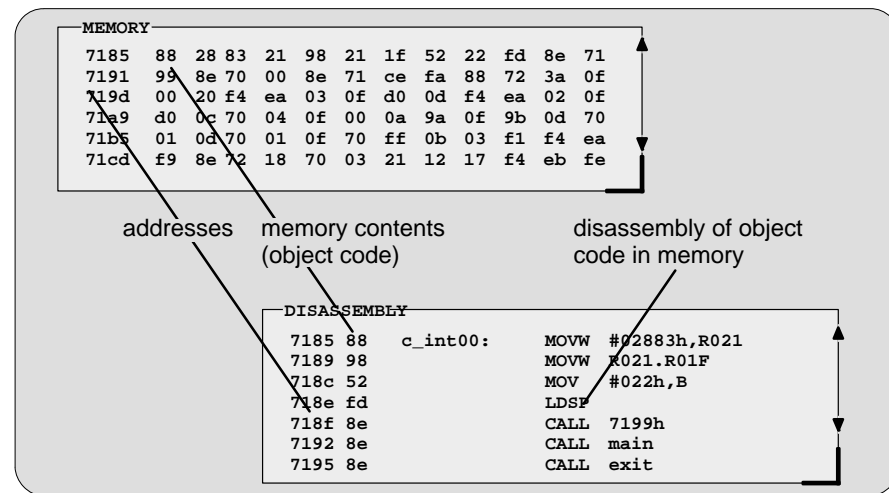
When you invoke the debugger, it comes up in auto mode. If you load an object file when you invoke the debugger, then the DISASSEMBLY window displays the reverse assembly of the object file that's loaded into memory. If you don't load an object file, the DISASSEMBLY window shows the reverse assembly 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** *label name*

This command modifies the display so that *address* or *label 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** *label name*

In assembly mode, ADDR works like the DASM command, positioning the code starting at *address* or at *label name* as the first line of code in the DISASSEMBLY window. In mixed mode, ADDR affects both the DISASSEMBLY and FILE windows.

To display assembly language code beginning with a local label inside a particular module, you would enter:

addr *file name.label name*

For example, assume you have a label called *loop* that is defined locally inside multiple modules. To display the code corresponding to loop in the file *example.asm*, you would enter:

addr **example.loop**

Modifying assembly language code

You can modify the code in the disassembly window on a statement-by-statement basis. The method for doing this is called *patch assembly*. Patch assembly provides a simple way to temporarily correct minor problems by allowing you to change individual statements and instruction words.

You can patch-assemble code by using a command or by using the mouse.

patch Use the PATCH command to identify the address of the statement you want to change and the new statement you want to use at that address. The format for this command is:

patch *address, assembly language statement*

For patch assembly, use the **right** mouse button instead of the left. (Clicking the left mouse button sets a software breakpoint.)

1) Point to the statement that you want to modify.

2) Click the right button. The debugger will open a dialog box so that you can enter the new statement. The address field will already be filled in; clicking on the statement defines the address. The statement field will already be filled in with the current statement at that address (this is useful when only minor edits are necessary).

Patch assembly may, at times, cause undesirable side effects:

❑ Patching a multiple-word instruction with an instruction of lesser length will leave "garbage" or an unwanted new instruction in the remaining old instruction fragment. This fragment must be patched with either a valid instruction or a NOP, or else unpredictable results may occur when you are running code.

❑ Substituting a larger instruction for a smaller one will partially or entirely overwrite the following instruction; you will lose the instruction and may be left with another fragment.

If you want to insert a large amount of new code or if you want to skip over a section of code, you can use a different patch assembly technique:

❑ To insert a large section of new code, patch a branch instruction to go to an area of memory not currently in use. Using the patch assembler, add new code to this area of memory, and branch back to the statement following the initial branch.

❑ To skip over a portion of code, patch a branch instruction to go beyond that section of code.

Effects of Patch Assembly

The patch assembler changes only the disassembled assembly language code—it does not change your source code. After determining the correct solution to problems in the disassembly, edit your source file, reassemble it, and reload the new object file into the debugger.

Additional information about modifying assembly language code

When you use patch assembly to modify code in the disassembly window, keep these things in mind:

❑ **Directives.** You cannot use directives (such as .global or .word).

❑ **Expressions.** You can use constants, but you cannot use arithmetic expressions. For example, an expression like 12 + 33 is not valid in patch assembly, but a constant such as 12 is allowed.

❑ **Labels.** You cannot define labels. For example, a statement such as the following is not allowed:

```
LOOP: BR LOOP
```

However, an instruction can refer to a label as long as it is defined in a COFF file that is already loaded.

❑ **Constants.** You can use hexadecimal, octal, decimal, and binary constants. The syntax to input constants is the same as that for the '370 assembler. (Refer to the *TMS370 Family Assembly Language Tools User's Guide.*)

❑ **Error messages.** The error messages for the patch assembler are the same as the corresponding '370 assembler error messages. Refer to the *TMS370 Family Assembly Language Tools User's Guide* for a detailed list of these messages.

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 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.

(Note that 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 7.3 on page 7-10.)

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 and FILE work similarly, but when you use FUNC, 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 ⒡⑨ to display the function; see the *CALLS window* discussion on page 4-9 for details.)

Displaying other text files

The DISASSEMBLY window always displays the reverse assembly of memory contents, regardless of 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. 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.

7.3 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 1.4, *Preparing Your Program for Debugging*, on page 1-11.)

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:

Emulator: Emulator using Windows: Application board:
   xds370 *object filename*   **xds370w** *object filename*   **abd370** *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:

Emulator: Application board:
   xds370 –s *object filename*   **abd370 –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*

If you don't supply an extension, the debugger will look for *filename*.out.

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*

If you enter the RELOAD command without specifying a filename, the debugger reloads the file that you loaded last.

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.

7.4 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, or

cd ■ 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 or Windows environment, you can name additional directories with the D_SRC environment variable. The format for doing this is:

SET D_SRC=pathname; 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 or initdb.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:

xds370 **–i** *pathname* [**–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.

7.5 Running Your Programs

To debug your programs, you must execute them on one of the two '370 debugging tools (emulator or application board). The debugger provides two basic types of commands to help you run your code:

❏ **Run commands** run your code on the target system without updating the display until you explicitly halt execution.

There are several ways to halt execution:

■ Set a breakpoint.
■ When you issue a run command, define a specific stopping point.
■ Press ESC.
■ Press 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 7-17).

❏ 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.

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:

XDS/22 emulator only

runb

Using the RUNB command to benchmark code is a multistep process, described later in this chapter (Section 7.7, *Benchmarking*, on page 7-20).

rrun   Use the RRUN (reset and run) command to reset the target system and begin program execution. This command is useful for verifying the initialization of the reset vector in your program. The format for this command is:

rrun [*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 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 7-17).

---

key

  F5   Pressing this key runs code from the current PC. This is similar to entering a RUN command without an *expression* parameter.

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.

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 7-17).

❑ 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 com-
cnext   mands. 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) Press and release the left mouse button.

To execute a NEXT,

1) Point to `Next=F10` in the menu bar.

2) Press and release the left mouse button.

Running code while connected to a target

reset   The RESET command resets the target system. The format for this command is:

reset

wrun   Use the WRUN (wait and run) command to wait for reset and run. The emulator will wait for a target system to ascert a reset, then it will run your program. The format for this command is:

wrun [*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 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 7-17).

Running code conditionally

The RUN, RRUN, WRUN, STEP, CSTEP, NEXT and CNEXT commands all 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; each time the debugger encounters a breakpoint, the expression is evaluated. 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.

Running code continuously

runf   Use the RUNF command to run the BTT independently from the CPU. The format for this command is:

runf

When you enter RUNF, the BTT and CPU begin execution simultaneously. With the CPU running, you can stop the BTT and perform operations such as dumping the contents of the trace buffer, updating the screen, and reconfiguring the BTT. To stop the BTT, press ESC. Otherwise, *you must wait for the BTT or CPU to stop on their own.* While both the BTT and CPU are running, you do not have access to the command line.

After you have finished operations on the BTT, you can restart it by entering the RUNF command again.

rrunf   Use the RRUNF command to reset the target system and begin execution of the BTT independently from the CPU. The format for this command is:

rrunf

XDS/22 emulator only

wrunf   Use the WRUNF command to wait for the target system to reset and then begin execution of the BTT independently from the CPU. The format for this command is:

wrunf

You can use the RRUNF and WRUNF commands to begin execution of the BTT and CPU initially. However, after you have halted the BTT by pressing ESC, you must restart the BTT by using the RUNF command.

halt   Use the HALT command to halt the CPU. 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 be running 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.

7.6   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.

[ESC]   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.

7.7 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*. This process is referred to as *benchmarking*.

Notes:

❑ The RUNB command will reconfigure the BTT setup; therefore, the previous setup is overwritten.

❑ The value in CLK is valid only after using a RUNB command that is terminated by a software breakpoint. (The maximum value for CLK is 65,535.)

❑ When programming in C, do not use a variable named CLK.

XDS/22 emulator only

Benchmarking code is a multiple-step process:

Step 1: Set the PC value at the statement that marks the beginning of the section of code you'd like to benchmark. (You can do this either by editing the PC value at the command line, or by setting a software breakpoint at the statement you'd like to benchmark and running to it.)

Step 2: Set a software breakpoint at the statement that marks the end of the section of code you'd like to benchmark.

Step 3: 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.

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

❑ '370 CPU registers

❑ Variables, including scalar types (ints, chars, etc.) and aggregate types (arrays, structures, etc.)

This chapter tells you how to display and change data.

8.1 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 windows** Display the contents of a range of memory, including the register file and peripheral file
CPU register values	**CPU window** Displays the contents of '370 CPU 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**.

8.2 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].a 43 giant[0].b –79 giant[0].c 19 etc.
? j	41
? j=0x5a	90
? i	–1
? i,x	0xff

The DISP command (described in detail on page 8-11) 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).

8.3　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.

↖ 　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 4.4, *The Active Window,* on page 4-21.)

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 additional data-management methods that take advantage of the fact that C expressions are accepted as parameters by most debugger commands, 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 A, you can enter:

```
? A
```

However, you can also use this type of command to modify A's contents. Here are some examples of how you might do this:
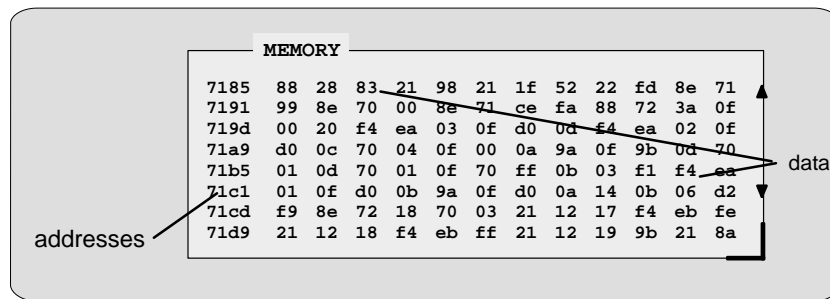
```
? A++              Side effect: increments the contents of A by 1
eval --A           Side effect: decrements the contents of A by 1
? A = 8            Side effect: sets A to 8
eval A/=2          Side effect: divides contents of A by 2
```

Note that not all expressions have side effects. For example, if you enter **? A+4**, the debugger displays the result of adding 4 to the contents of A but does not modify A'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:

=	+=	−=	*=	/=
%=	&=	^=	|=	<<=
>>=	++	− −		

8.4   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 windows* discussion (page 4-14).


```
        ┌─ MEMORY ─────────────────────────────────┐
        │ 7185  88 28 83 21 98 21 1f 52 22 fd 8e 71 │
        │ 7191  99 8e 70 00 8e 71 ce fa 88 72 3a 0f │
        │ 719d  00 20 f4 ea 03 0f d0 0d f4 ea 02 0f │
        │ 71a9  d0 0c 70 04 0f 00 0a 9a 0f 9b 0d 70 │  data
        │ 71b5  01 0d 70 01 0f 70 ff 0b 03 f1 f4 ea │
        │ 71c1  01 0f d0 0b 9a 0f d0 0a 14 0b 06 d2 │
        │ 71cd  f9 8e 72 18 70 03 21 12 17 f4 eb fe │
        │ 71d9  21 12 18 f4 eb ff 21 12 19 9b 21 8a │
        └──────────────────────────────────────────┘
   addresses
```

The debugger has commands that show the data values at a specific location or that display a different range of memory in the MEMORY window. The debugger allows you to change the values at individual locations; refer to Section 8.3 (page 8-4) for more information.

Displaying memory contents

The main way to observe memory contents is to view the display in a MEMORY window. Four MEMORY windows are available: the default window is labeled MEMORY, and the three additional windows are called MEMORY1, MEMORY2, and MEMORY3. Notice the default window does not have an extension number in its name; this is because MEMORY1, MEMORY2, and MEMORY3 are pop-up windows that can be opened and closed throughout your debugging session. Having four windows allows you to view four different memory ranges.

The amount of memory that you can display is limited by the size of the individual MEMORY windows (which is limited only by the screen size). During a debugging session, you may need to display different areas of memory within a 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 syntax for this command is:

mem   *expression* [*, display format*]

To view different memory locations in an additional MEMORY window, use the
MEM command with the appropriate extension number on the end. For exam-
ple:

To do this. . .	Enter this. . .
View the block of memory starting at address 0x8000 in the MEMORY1 window	**mem1** 0x8000
View the same block of memory (starting at ad-dress 0x8000) but in the MEMORY2 window	**mem2** 0x8000

> **Note:**
>
> If you want to view a different block of memory explicitly in the default
> MEMORY window, you can use the aliased command, MEM0. This works
> *exactly* the same as the MEM command. To use this command, enter:
>
> **mem0** *address*

For more information, see the *MEMORY windows* discussion on page 4-14.

The *expression* you type in represents the address of 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; to show fewer loca-
tions, make the window smaller. (See *Resizing a window*, page 4-24, for more
information.)

Expression can be an absolute address, a symbolic address, or any C expres-
sion. 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 0x0000

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**; otherwise, the debugger treats the number as a decimal address.

❑ **Symbolic address.** You can use any defined C symbol. 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 – A+ 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 4-29) 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 26 (hex), you could enter:

? *0x26

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 *0x26

❑ 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 *)0x26

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 *TMS370 Family Assembly Language 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 of the block in bytes. This parameter can be any C expression.

❑ The *filename* is a system file.

  If you don't supply an extension, the debugger adds an .obj extension.

For example, to save the values in data memory locations 0x0–0x10 to a file named memsave, enter:

```
ms 0x0,1,0x10,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.obj ⏎
```

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 words to fill.
❑ The *data* parameter is the value that is placed in each word in the block.

For example, to fill locations 0x8000 to 0x8003 with the value 0x12, enter:

```
fill 0x8000,0x4,0x12 ⏎
```

If you want to check to see that memory has been filled as you have asked, you can enter:

```
mem 0x8000 ⏎
```

This changes the MEMORY window display to show the block of memory beginning at address 0x8000.

Note that the FILL command can also be executed from the Memory pulldown menu.

8.5 Managing Register Data

In mixed and assembly modes, the debugger maintains a CPU window that displays the contents of individual registers. For details concerning the CPU window, see the *CPU window* discussion (page 4-17).

```
 CPU
  PC    7185
  A     87
  B     00
  ST    40
  SP    22
```

register name → ... ← register contents
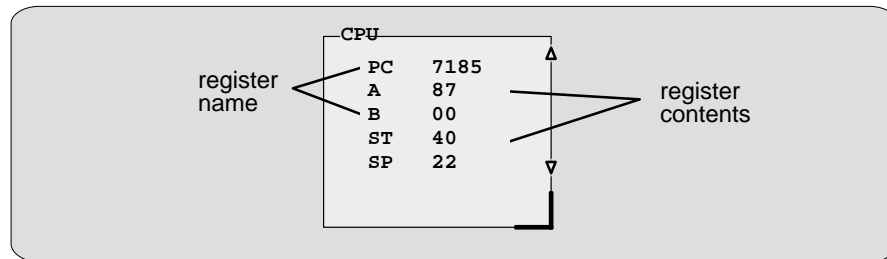
The debugger provides commands that allow you to display and modify the contents of specific registers. Remember, you can use the data-management commands or the debugger's overwrite editing capability to modify the contents of any register displayed in the CPU or WATCH window. Refer to Section 8.3 (page 8-4) 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 two 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, enter:

   ? SP

   The debugger displays the SP's current contents in the COMMAND window display area.

❑ If you want the opportunity to observe a register over a longer period of time, you can display it in a WATCH window. Use the WA command to do this. 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.

When you're debugging C in auto mode, these methods are also useful because the debugger doesn't show the CPU window in the C-only display.

8.6   Managing Data in a DISP (Display) Window

The main purpose of the DISP window is to display the values of 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 4-18).
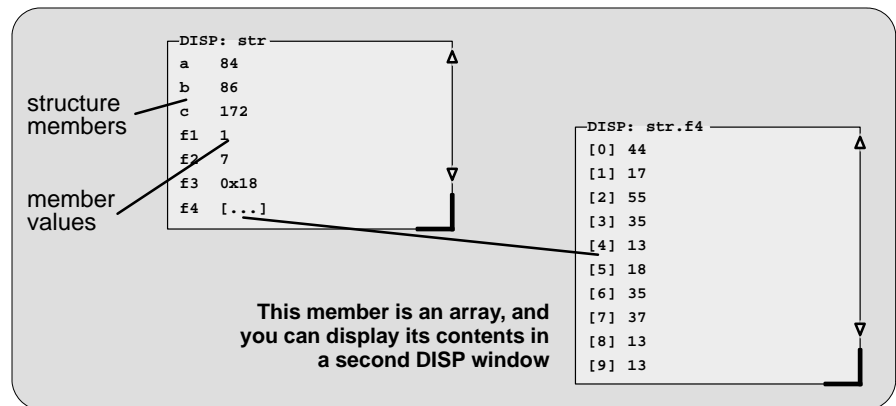
```
          ┌─DISP: str───────────────────┐
          │ a    84                     △│
          │ b    86                     │
structure │ c    172                    │        ┌─DISP: str.f4────────────┐
members ──│ f1   1                      │        │ [0] 44                △│
          │ f2   7                      │        │ [1] 17                  │
          │ f3   0x18                   │        │ [2] 55                  │
member   ─│ f4   [...]                  │        │ [3] 35                  │
values    │                            ▽│        │ [4] 13                  │
          └─────────────────────────────┘        │ [5] 18                  │
                                                  │ [6] 35                  │
                  This member is an array, and    │ [7] 37                  │
                  you can display its contents in  │ [8] 13                ▽│
                  a second DISP window             │ [9] 13                  │
                                                  └─────────────────────────┘
```

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 8.3 (page 8-4), 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* [*, display format*]

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   0x0000

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 4.4, *The Active Window*, on page 4-21).

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 all DISP windows when you execute a LOAD or SLOAD command.

8.7 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.
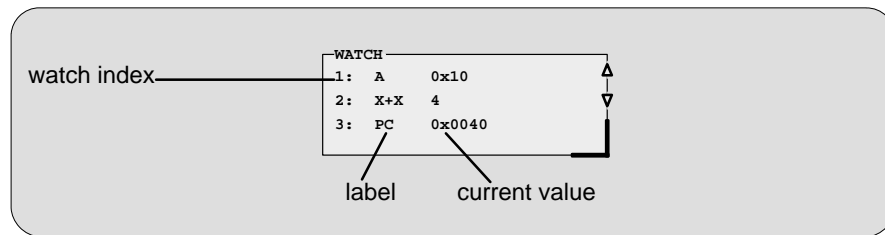
The debugger displays a WATCH window *only when you specifically request a WATCH window* with the WA command (described below). Note that there is only one WATCH window. For additional details concerning the WATCH window, see the *WATCH window* discussion (page 4-19).

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 8.3 (page 8-4), for more information.

Note:

All of the watch commands described can also be accessed from the Watch pulldown menu. For more information about using the the pulldown menus, refer to Section 5.2, *Using the Menu Bar and the Pulldown Menus* (page 5-7).

```
Watch
Add
Delete
Reset
```
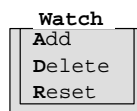
Displaying data in the WATCH window

The debugger has one command that you can use to add items to the WATCH window.

wa   To open the WATCH window, use the WA (watch add) command. The basic syntax is:

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

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 (such as i++). In an assembly language program, you can use a symbol name as the expression parameter. To watch the contents of a symbol name, precede it with an asterisk:

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

If you want to watch a symbol that is local to a particular module, you must include the module name in the expression. For example, to watch the contents of *symbol_name*, a local symbol to the module *sample.asm*, you would enter:

wa *sample.symbol_name [,[*label*], *display format*]

Note:

To watch a symbol that is local to a particular module, give an extension of .c or .asm to the module name.

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 8-14 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 re-maining 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.

8.8   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*. The following is a list of available data formats:

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**

Only a subset of the display formats applies to each data type. Table 8–1 lists the C data types that can be used for the *data type* parameter, and shows valid combinations of data types and display formats.

Table 8–1. Data Types for Displaying Debugger Data

	Valid Display Formats									
Data Type	**c**	**d**	**o**	**x**	**e**	**f**	**p**	**s**	**u**	**Default Display Format**
char	√	√	√	√					√	ASCII (c)
uchar	√	√	√	√					√	Unsigned (u)
short	√	√	√	√					√	Decimal (d)
ushort	√	√	√	√					√	Unsigned (u)
int	√	√	√	√					√	Decimal (d)
uint	√	√	√	√					√	Unsigned (u)
long	√	√	√	√					√	Decimal (d)
ulong	√	√	√	√					√	Unsigned (u)
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 unsigned decimals, 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:   Unsigned decimal
Type int:             Decimal
Type unsigned int:    Unsigned decimal
Type short:           Decimal
Type unsigned short:  Unsigned decimal
Type long:            Decimal
Type unsigned long:   Unsigned decimal
Type float:           Exponential floating point
Type double:          Exponential floating point
Type pointer:         Hexadecimal
```

❑ 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. However, there is one exception: you cannot use the *s* display format parameter with the MEM command.

Using Software Breakpoints

This chapter describes the simple processes of setting and clearing software breakpoints and of obtaining a listing of all the breakpoints that are set.

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 software breakpoints at critical points in your code. You can set these breakpoints in assembly language code and in C code. A software breakpoint halts any program execution, whether you're running or single-stepping through code.

9.1 Setting a Software Breakpoint

When you set a software breakpoint, the debugger highlights the breakpointed line in two ways:

❏ It prefixes the statement with the > character.

❏ 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──────────────────────────────┐
 00044
 00045  >        meminit();
 00046          for (i=0; i < 0x70;i++)
 00047          {
 00048              call(i);

 ┌─DISASSEMBLY─────────────────────────────────┐
 7028 8e           MOV    A,Z(ROLF)
 702C 8e >         CALL   meminit
 702f fa           CLR    A
```
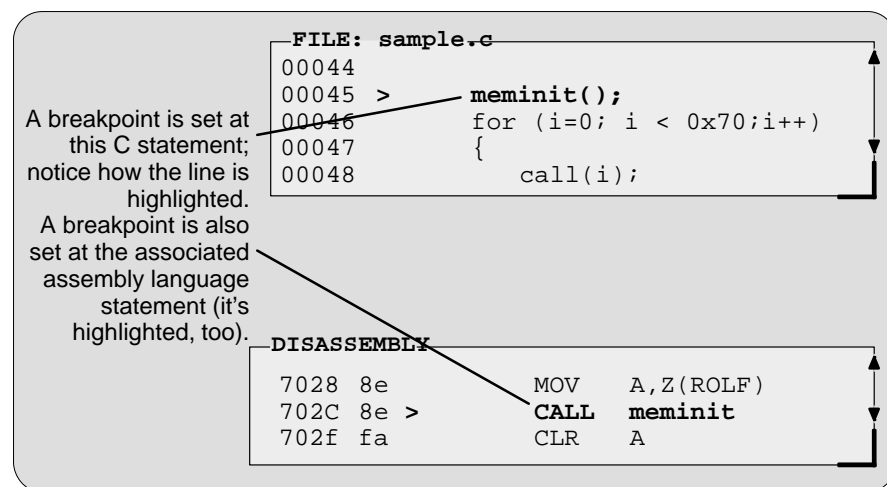
Notes:

❏ 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 software breakpoints can be set.

❏ During program execution, the debugger executes a NOP statement after encountering *each* software breakpoint. Because it takes seven clock cycles to execute a NOP statement, the '370 device timers and the corresponding prescalar are incremented by seven.

There are several ways to set a software breakpoint:

1) Make the FILE or DISASSEMBLY window the active window.

2) Point to the line of assembly language code or C code where you'd like to set a breakpoint.

3) 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.

3) Press the F9 key.

Repeating this action clears the breakpoint.

ba If you know the address where you'd like to set a software breakpoint, you can use the BA 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.

9.2 Clearing a Software Breakpoint

There are several ways to clear a software breakpoint. If you clear a breakpoint from an assembly language statement, the breakpoint is also cleared from any associated C statement; if you clear a breakpoint from a C statement, the breakpoint is also 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 break-pointed assembly language or C statement.

2) Press the F9 key.

br   If you want to clear **all** the software breakpoints that are set, use the BR command. This command is useful because it doesn't require you to search through code to find the desired line. The syntax for the BR command is:

br

bd   If you'd like to clear one specific software breakpoint and you know the address of this breakpoint, you can use the BD 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.

9.3 Finding the Software Breakpoints That Are Set

bl Sometimes you may need to know where software breakpoints are set. For example, the BD command's *address* parameter must correspond to the address of a breakpoint that is set. The BL command provides an easy way to get a complete listing of all the software breakpoints that are currently set in your program. The syntax for this command is:

bl

The BL command displays a table of software breakpoints in the COMMAND window display area. BL lists all the software breakpoints that are set, in the order in which you set them. Here's an example of this type of list:

```
 Address     Symbolic Information
4000
7000      in main, at line 45, "c:\370tools\sample.c"
5000
```

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, the way 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.

10.1 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, followed by 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 10–1 lists the valid values for the *attribute* parameters.

Table 10–1. Colors and Other Attributes for the COLOR and SCOLOR Commands

(a) Colors

black	blue	green	cyan
red	magenta	brown	white

(b) Other attribute

bright

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 10–2 lists valid values for the *area name* parameters. This is a long list; the subsections following the table further identify these areas.

Table 10–2. 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 10–2 (left to right, top to bottom).

The remainder of this section identifies these areas.

Area names: common display areas
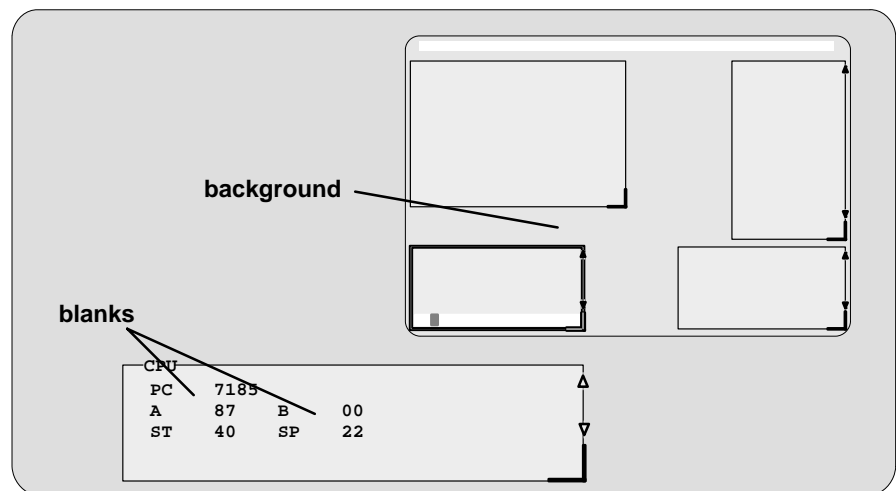
Area identification	Parameter name
Screen background (behind all windows)	background
Window background (inside windows)	blanks

Area names: window borders
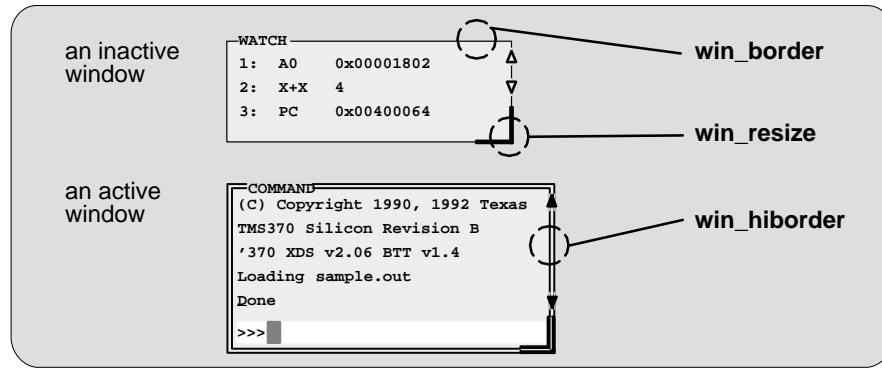
Area identification	Parameter name
Window border for any window that isn't active	win_border
The reversed "L" in the lower right corner of a resizable window	win_resize
Window border of the active window	win_hiborder

Area names: COMMAND window
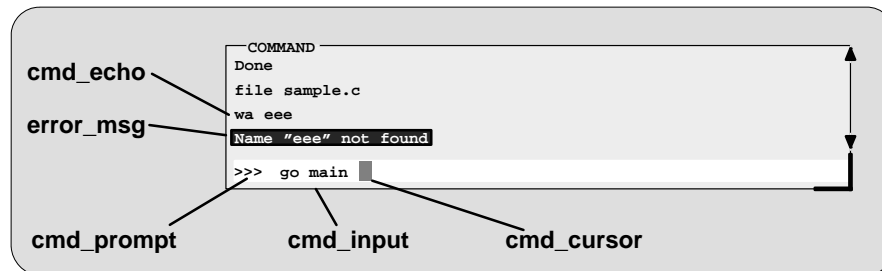
Area identification	Parameter name
Echoed commands in display area	cmd_echo
Errors shown in display area	error_msg
Command-line prompt	cmd_prompt
Text that you enter on the command line	cmd_input
Command-line cursor	cmd_cursor

Area names: DISASSEMBLY and FILE windows

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
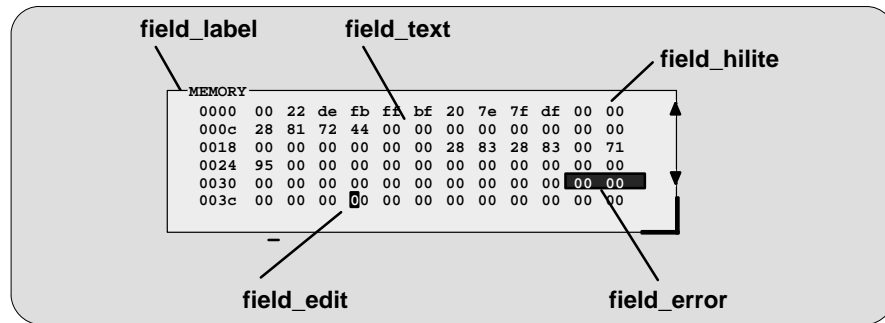
Area identification	Parameter name
Label of a window field (includes register names in CPU window, addresses in MEMORY window, index numbers and labels in WATCH window, member names in DISP window)	field_label
Text of a window field (includes data values for all data-display windows) and of most command 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
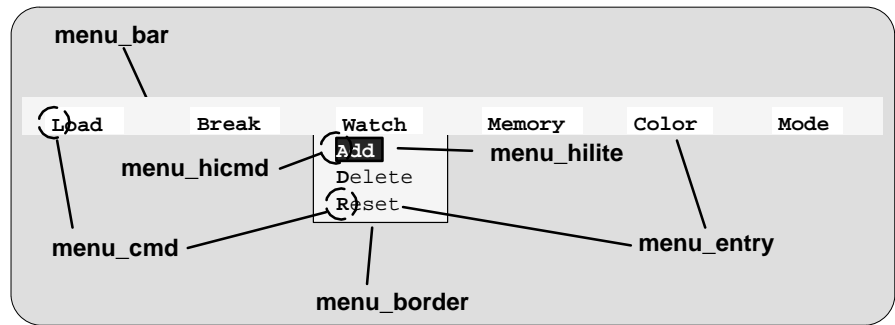
Area identification	Parameter name
Top line of display screen; background to main menu choices	menu_bar
Border of any pulldown menu	menu_border
Text of a menu entry	menu_entry
Invocation key for a menu or menu entry	menu_cmd
Text for current (selected) menu entry	menu_hilite
Invocation key for current (selected) menu entry	menu_hicmd

10.2 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. Note that you can skip parameters, if desired.

```
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
```

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*.

10.3 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 called init.clr. The screen configuration file defines how various areas of the display will appear. If the debugger doesn't find this file, it uses the default screen 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.

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 resolution, border styles, colors, window positions, window sizes, and (on PCs) video mode (EGA, VGA, CGA, etc.) 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 specify path information, the debugger places the file in the current directory. If you don't supply a filename, the debugger saves the current configuration into a file named init.clr.

Note that 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 resolution, colors, window positions, window sizes, border styles, and (on PCs) video mode (EGA, CGA, MDA, etc.) saved in *filename*. Screen resolution and video mode are restored either by changing the mode (on video cards with switchable modes) or by resizing the debugger screen (on other hosts).

If you don't supply a *filename*, the debugger looks for init.clr. The debugger searches for the file in the current directory and then in directories named with the D_DIR environment variable.

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 initialization batch file that the debugger executes at invocation time (init.cmd). 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.

10.4 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.)

Note that 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 initialization batch file that the debugger executes at invocation time (init.cmd).

You can also execute this command as the Prompt selection on the Color pull-down menu.

Using Hardware Breakpoint, Trace, and Timing Features

This chapter tells you how to use the features of the BTT (breakpoint/trace/timing) board. The BTT is a separate board that is included as part of your XDS/22 emulation system. The BTT monitors the CPU; when a preselected pattern of bus activity is detected, the BTT performs an action such as executing a hardware breakpoint or storing information in the trace buffer.

Unlike most debugger features, many BTT features are accessed not with commands but only with a detailed set of menus and dialog boxes.

11.1 Running a BTT Session

To use the BTT, you will need to decide which types of actions you would like the BTT to perform and under what conditions you want these actions to be performed. Once you have defined actions and made any necessary global changes, you can run your program. If you are collecting traces or timing information, you can view the collected information.

Step 1: **Define actions for up to four states.** The BTT monitors the '370 device, watching for selected types of bus activity. The BTT can look for four different combinations of bus activity. These combinations are referred to as *states* and are labeled as state 0 through state 3. Each state can define one or more of the following actions:

❏ **Breakpoint/event.** The BTT can perform a hardware breakpoint, which halts both the CPU and the BTT, or decrement one of several counters.

❏ **Trace.** The BTT can store a trace sample (a collection of information about address and bus contents, memory cycles, timing, and opcodes) in the trace buffer.

❏ **Jump.** The BTT can jump to one of the other three states.

❏ **Start or stop the point or range timer.** The BTT can use the point or range timer to perform timing analysis.

You can define actions for as many of the four states as you need to use. The BTT uses the states sequentially, beginning with state 0.

An action can take place only when certain conditions (which you define for individual actions) take place. The matching of bus activity to the conditions that you defined is called **qualifying**. The conditions that you can define to qualify an action include:

❏ **Address accesses.** Actions can be qualified when a specific address is accessed, when an address is accessed inside of a range, or when an address is accessed outside of a range.

❏ **Data accesses**. Actions can be qualified when a specific data value is accessed, when a value is accessed inside of a range of values, or when an address is accessed outside of a range of values.

❏ **Memory cycles.** Actions can be qualified on reads, writes, instruction acquisitions, or any combination of these memory cycles.

❏ **External signals.** The BTT has an external-signal probe that can be connected to a maximum of eight external signals. Actions can be qualified on the logic levels on these lines.

Conditions can be combined; for example, you could define a condition that would tell the BTT to look for a write of a certain value to any address within a range.

For more information about these topics, see Section 11.2, *Accessing Essential BTT Features*, page 11-4, and Section 11.3, *Defining Conditions for an Action,* page 11-6.

Step 2: **Enter appropriate global settings.** You can change certain aspects of basic BTT-board operation through the global settings. Note that the default global settings are sufficient for many applications, so you may not find it necessary to change the global settings.

These components are controlled through global settings:

❏ The **delay counter** defines how many additional trace samples should be collected before a breakpoint is executed.

❏ The **max trace** defines the maximum number of trace samples to collect.

❏ The **end state** defines the last state in a sequence of states.

❏ The **loop counter** defines how many times the BTT will sequence through states before executing a breakpoint.

❏ The **time-out timer** defines how long your program can run.

Step 3: **Run your program.** Once you have defined actions for as many states as you wish, you should run your program. Use any of the run commands—RUN, STEP, etc. Whenever program execution causes bus activity to match any conditions that you have defined, the BTT will perform the appropriate action. Your program will continue running until you explicitly halt it, until it reaches a software breakpoint, until it has run for length of time defined by the time-out timer, or until a hardware breakpoint is qualified.

Step 4: **View the trace buffer and timing information.** If you are collecting trace samples or are using the point or range timer, you can view the trace buffer and/or the timing statistics through the INSPECT window. For more information about this topic, see Section 11.9, *Viewing Trace Buffer and Timing Information*, page 11-20.

11.2 Accessing Essential BTT Features

Many of the basic tasks that you will need to carry out—viewing information about specific states, adding an action to a state, deleting an action from a state, and accessing global settings—can be accomplished through a single dialog box called the BTT Setup dialog box.

To open the BTT Setup dialog box, select Setup from the BTT menu. This opens a dialog box like the one shown in Figure 11–1.

Figure 11–1. The BTT Setup Dialog Box

```
┌─BTT Setup─────────────────────────────────────────────────────┐
│ State 0        <<Next state>>  <ClEar state>   <Add action...> │
│                                                                │
│ Action: BP/event    Action: Trace                             │
│ addr = 7050         addr = 7000                               │
│ data = IGNORE            ...7FFF                              │
│  cycles ALL         data = IGNORE                            │
│   extern IGNORE       cycles ALL                             │
│                       extern IGNORE                          │
│  ┌─State Mode──────┐  ┌─Trace Mode──┐  ┌──────────────────┐  │
│  ( ) Address only    (*) Normal      Event count [1 ......... ] │
│  (*) address and data ( ) TRIX                               │
│  └─────────────────┘  └─────────────┘  └──────────────────┘  │
│                                                                │
│ <Load...>  <Store...> <Globals...>            <OK> <Cancel>    │
└────────────────────────────────────────────────────────────────┘
```

Current state (→ State 0)

Descriptions of actions defined for the current state
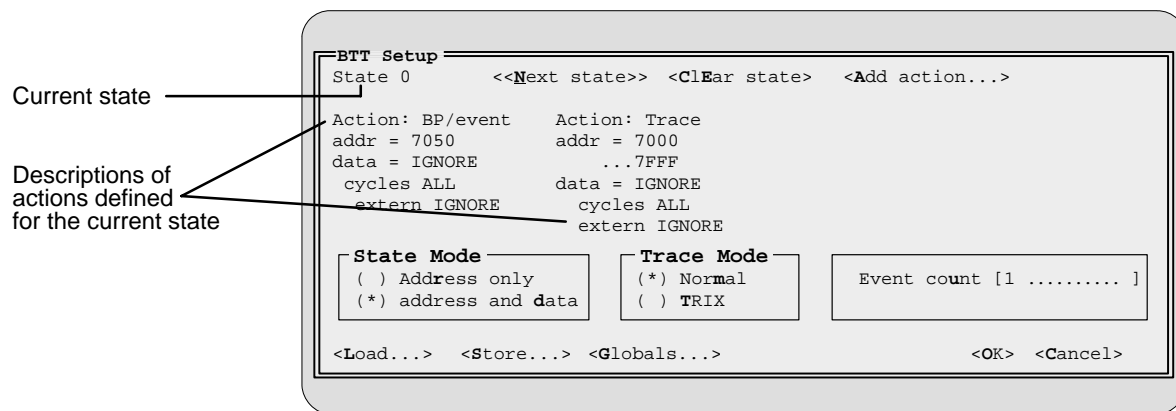
You can use the BTT Setup dialog box to perform these tasks:

❏ **View information about states.** The BTT Setup dialog box shows which state is current and which actions are defined for that state. For example, in Figure 11–1, state 0 is the current state; a BP/event (breakpoint/event) and a trace are defined as actions for state 0.

❏ **Select the current state.** Figure 11–1 shows settings for state 0. To get to state 1, click on <Next state>. Clicking again on <Next state> will take you to state 2, then state 3, and back to state 0 again.

❏ **Reset the current state.** If you want to clear out all actions for a particular state, make it the current state, then click on <Clear state>. This deletes all actions defined for the current state.

❏ **Add an action to the current state.** To add an action to the current state, click on <Add action...>. The debugger will display a menu so that you can select the type of action you want to add. When you select an action from the menu, the debugger will display another dialog box (referred to as an *action dialog box*) where you can define the conditions that will qualify the action. For more information, see Section 11.3, *Defining Conditions for an Action*, on page 11-6.

❑ **Edit an existing action.** To make changes to the conditions that you have defined for an existing action, click on the action description. The debugger will display the associated action dialog box. Make your changes, then click on <OK>.

❑ **Delete an action from the current state.** Deleting an action is similar to editing an action. Click on the action you want to delete. The debugger will display the associated action dialog box; click on <Delete>. This deletes the action from the current state.

❑ **Select the state mode.** The state mode helps to define the types of conditions that can apply to actions. You can choose to qualify actions on address accesses only or on a combination of address and data accesses. To do this, click on the appropriate field in the State Mode box. By default, you can qualify an action on a combination of both address and data accesses. Note that address-and-data state mode can limit the number of actions that you can define for a state; for more information, see Section 11.4, *Limits on the Number of Actions per State,* on page 11-11.

❑ **Select the trace mode.** The trace mode helps to define what types of memory cycles will be stored in the trace buffer. For more information about tracing and the trace mode, refer to Section 11.7, *Collecting Trace Samples*, on page 11-17.

❑ **Set the event counter.** The event counter is used with several global components to determine when a hardware breakpoint should occur. If you leave the event counter at its default setting (1), a hardware breakpoint can occur as soon as a BP/event is qualified. For more information about breakpoints and the event counter, refer to Section 11.6, *Using Hardware Breakpoints and Events*, on page 11-14.

❑ **Load/store a setup.** You can easily save your BTT configuration and use it again for another session. For more information, refer to Section 11.10, *Reusing a BTT Setup*, on page 11-24.

❑ **Alter basic board operation.** You can change certain aspects of basic BTT-board operation through the global settings. The global settings are described throughout this chapter—for example, the time-out timer is described with the timing analysis information.

❑ **Confirm or suspend the BTT setup and close the dialog box.** When you have satisfactorily completed all of your selections for each state, click on <OK>. If at any time you want to discard all of your selections for the action, click on <Cancel>.

11.3 Defining Conditions for an Action

To add an action to a state, bring up the BTT Setup dialog box and click on
<Add action...>. You'll see a Select action menu like the one in Figure 11–2.

Figure 11–2. The Select Action Menu

```
╒═ Select action ═══╕
│<<Bp/Event...>>   2│
│ <Trace...>       2│
│ <Jump...>        1│
│ <RangeTimer...> 1 │
│ <PointTimer...> 2 │
│          <Cancel> │
└───────────────────┘
```
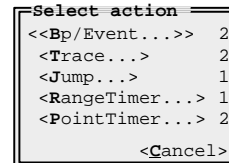
This menu contains two types of information:

❑ It contains fields that allow you to choose whether you want to define
conditions for a hardware breakpoint/event, a trace sample, a jump to a
new state, or timing analysis.

❑ It shows you the number of specific actions that can be defined for the
state. (The number of actions you can define depends on which state
mode you've chosen and on the number and type of actions that are
already defined for a state. For more information, see Section 11.4, *Limits
on the Number of Actions Per State*, on page 11-11.)

When you choose one of the items in the menu, you'll see a dialog box similar
to the one shown in Figure 11–3.

Figure 11–3. The Dialog Box for Defining Conditions

```
╒═ BP/Event action ══════════════════════════════════════════════╕
│                             Jump to [N/A]                       │
│  ┌─ Address qualifiers ─────────────────────────────────────┐   │
│  │ (*)One point    ( )Two points    ( )In range    ( )Outside range │
│  └──────────────────────────────────────────────────────────┘   │
│ addr1=[0x0000 ................................................. ]│
│ addr2=[N/A                                                      ]│
│  mask [0x0000 ......... ]                                        │
│  ┌─ Data qualifiers ────────────────────────────────────────┐   │
│  │ (*)One point    ( )Two points    ( )In range    ( )Outside range │
│  └──────────────────────────────────────────────────────────┘   │
│ data1=[0x00 ................................................... ]│
│ data2=[N/A                                                      ]│
│  mask [0x00 ........... ]                                        │
│                                  extern [0x00 ................ ] │
│  ┌─ Cycle ───────────────┐        mask [0x00 ....... ]           │
│  │ [X] MR   [X] MW   [X] IAQ │                                   │
│  └───────────────────────┘     <<OK>>   <Cancel>  <Delete>       │
└─────────────────────────────────────────────────────────────────┘
```
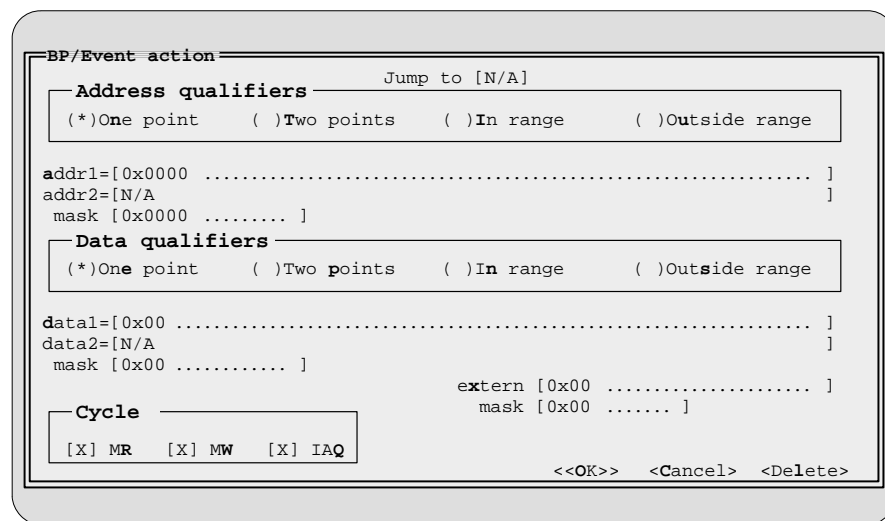
This dialog box is used for qualifying all actions, with minor variations. For example, the dialog box in Figure 11–3 is labeled BP/Event action; if you select a different action, the label changes to show the action you selected. (For this reason, this type of dialog box is referred to as an *action dialog box.*)

Some of the fields in an action dialog box may not apply to each action, even though they are shown in each version of the dialog box. When this is the case, the field is lowlighted (grayed out) and the debugger prevents you from accessing the field (you can't click on the field, tab to it, etc.).

Like menus and other dialog boxes, the action dialog box has highlighted characters that enable you to select a field by pressing ⒜⒧⒯ and the highlighted character key. Some fields in the action dialog box do not have highlighted characters. To select them without using the mouse, use ⒜⒧⒯ with the appropriate key to select the nearest field, then use ⒯⒜⒝ or ⟶ to reach the field you want to edit.

When you are finished defining the conditions for an action, click on <OK>. This adds the action to the current state. If you want to delete an action from the current state, click on <Delete>. Clicking on <Cancel> adds the action to the state without saving the conditions that you defined.

Defining a jump

When you select Jump from the Select action menu, the label at the top of the action dialog box changes from:

```
Jump to [N/A]
```

to:

```
Jump to [0...]
```

This happens so that you can select another state to jump to when the conditions for the jump (defined by the address, data, and cycle qualifiers) are met.

Defining address qualifiers

You can qualify any action based on the contents of the address bus. Address conditions are defined in this portion of the action dialog box:

```
┌ Address qualifiers ─────────────────────────────────────────┐
│(*)One point    ( )Two points ( )In range    ( )Outside range│
└─────────────────────────────────────────────────────────────┘

 addr1=[0x0000 ......................................... ]
 addr2=[N/A                                              ]
  mask [0x0000 .. ]
```

You can use addresses in four ways:

❑ **Qualify on a single address.** When you select **One point** as the address qualifier, you can enter a single address to be used as part of the condition for the action. You enter the address in the addr1 field; the addr2 field will be lowlighted because you can't access it when you select One point.

This is the default setting.

❑ **Qualify on one of two separate addresses.** When you select **Two points**, the action can take place when either of the addresses is accessed. You enter the first address in the addr1 field and the second address in the addr2 field. The contents of the addr2 field will change from N/A to 0x0000 so that you can enter an address. (To get to the second address field, use the mouse or press the ⌷TAB⌷ key.)

❑ **Qualify on an address within a range.** Select **In range** to define an inclusive range. The action can take place when any address in the range is accessed. Enter the beginning address in the addr1 field and the ending address in the addr2 field.

❑ **Qualify on an address outside of a range.** Select **Outside range** to define an exclusive range. The action can take place when:

■ An address is accessed that is lower than the address defined by addr1, or

■ An address is accessed that is higher than the address defined in the addr2 field.

To get to the addr2 field, use the mouse, press ⌷TAB⌷, or press ⌷→⌷.

Additionally, you can **mask** addresses, which means that specific address bits will be ignored; refer to *Masking qualifiers*, page 11-10.

Defining data qualifiers

The options for defining data qualifiers are similar to the options for defining address qualifiers—you can define conditions on a single data point, a pair of points, an exclusive range, or an inclusive range. Data qualifiers are defined in this portion of the dialog box:

```
┌ Data qualifiers ───────────────────────────────────────┐
│(*)One point    ( )Two points ( )In range    ( )Outside range│
└─────────────────────────────────────────────────────────┘

 data1=[0x00 ................................................ ]
 data2=[N/A                                                   ]
  mask [0x00 ..... ]
```

Whether or not you can use data qualifiers as part of the condition depends on whether you select address-only or address-and-data state mode. (For more information about state modes, see Section 11.4, *Limits on the Number of Actions per State*, on page 11-11.) If you select address-only state mode, you can use address qualifiers but not data qualifiers; if you select address-and-data state mode, then you can use data qualifiers with or without address qualifiers.

To get to the data2 field, use the mouse, press ⌷TAB⌷ , or press ⌷→⌷.

Data values can be masked so that specific bits within the value are ignored; see *Masking qualifiers* on page 11-10.

Defining external-signal qualifiers

The BTT has an external-signal probe that can be connected to eight external signals. You can qualify actions based on the logic levels on these signal lines. The external-signal qualifier is defined in this portion of the dialog box:

```
            extern [0x00 ................... ]
              mask [0x00 .... ]
```

The extern field defines the pattern of activity that you want to use for qualifying the action. A 1 indicates that you are qualifying a high level on a signal, and a 0 indicates that you are qualifying a low level on a signal. For more information about using the external probes, refer to the installation instructions.

The external-signal value can be masked so that specific signal bits are ignored; see *Masking qualifiers* on page 11-10.

Masking qualifiers

Masking is a method of indicating don't-care bits in an address, data, or external-signal value. Address qualifiers use a 16-bit mask; data and external-signal qualifiers use an 8-bit mask. A 0 bit in a mask causes the corresponding bit in the address, data, or external-signal qualifier to be ignored.

In the dialog boxes, the qualifier and mask values initially come up as 0s. As soon as you enter an address, data, or external-signal qualifier, the associated mask value immediately changes to 0xFFFF (for addresses) or 0xFF (for data and the external signals). This is the same as a binary value of all 1s; because there are no don't-care bits, the mask is essentially prevented from affecting the qualifier value. (To get to the mask field, use the mouse, press ⌷TAB⌷, or press ⤷.)

If you want to ignore bits in a qualifier value, enter a new mask value. For example, suppose you want to qualify some action on any address that ends in 80h. To do this, specify a mask value of 0x00FF and any address whose last two digits are 0x80 (128_{10}):

Address value:	1 0 1 0 1 0 0 1 1 0 0 0 0 0 0 0	(0xA980)
Mask value:	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1	(0x00FF)
Qualifying addresses:	X X X X X X X X 1 0 0 0 0 0 0 0	(0xXX80)

The eight MSBs of this mask value are 0s, which ensures that the eight MSBs of any address will be ignored. The eight LSBs of the mask value are 1s, ensuring that the only addresses that can qualify the action are those whose eight LSBs match those of the address that you supplied.

Defining cycle qualifiers

You can qualify an action on a specific memory cycle. Cycle qualifiers are defined in this part of the dialog box:

```
┌ Cycle ─────────────────┐
│ [X] MR    [X] MW    [X] IAQ │
```

You can choose one or more of the following cycles:

❑ **Memory read.** To qualify an action on a memory read, select **MR**.

❑ **Memory write.** To qualify an action on a memory write, select **MW.**

❑ **Instruction acquisition.** To qualify an action on an instruction acquisition, select **IAQ.**

By default, all three memory cycles are enabled as part of the condition for the action.

11.4 Limits on the Number of Actions per State

You can define more than one action per state. For example, you may want to trace accesses within a certain range of memory, then breakpoint when an address outside of the range is accessed. Such a combination (tracing and breakpointing) can be associated with a single state. However, the BTT limits the total number of actions that you can define per state. The basic limits depend on which state mode you've selected:

❑ **Address only—four actions maximum.** In address-only state mode, you can qualify actions by using address values but not data values. Because you're not using data values, some of the BTT resources are freed. As a result, you can define a maximum of four actions for the state.

❑ **Address and data—two actions maximum.** In address-and-data state mode, you can qualify actions by using both address and data values. This uses more of the BTT resources. As a result, you can define a maximum of two actions for the state.

The state mode is defined in this part of the BTT Setup dialog box:

```
┌ State Mode ─────────┐
( ) Address only
(*) address and data
└─────────────────────┘
```

Each state has its own state mode; for example, state 0 could use address-only state mode, and state 1 could use address-and-data state mode.

Limitations apply not only to the number of actions, but to the type and combination of actions. For example, in address-only state mode, although you may be able to select four actions, you can't select four Jump actions—you can select only two.

Figure 11–4 *(a)* shows the initial Select action menu for address-only state mode, listing the number of actions that can currently be defined for the state. Figure 11–4 *(b)* shows how the number of available actions changes after a BP/event is defined.

Figure 11–4. How the Select Action Menu Changes After Actions Are Defined

(a) Initial menu (address only)   *(b) Menu after a BP/event is defined*
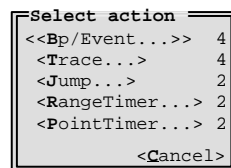
```
┌═Select action ═══┐          ┌═Select action ═══┐
│<<Bp/Event...>>  4│          │<<Bp/Event...>>  3│
│ <Trace...>      4│          │ <Trace...>      3│
│ <Jump...>       2│          │ <Jump...>       2│
│ <RangeTimer...> 2│          │ <RangeTimer...> 1│
│ <PointTimer...> 2│          │ <PointTimer...> 2│
│        <Cancel>  │          │        <Cancel>  │
└══════════════════┘          └══════════════════┘
```
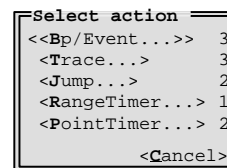
Table 11–1 summarizes the limit on the number and combination of actions. For example, the first several lines in Table 11–1 *(a)* show that in address-only state mode, you could, for a single state, define four BP/events, or three BP/events and one trace, or three BP/events and one jump, etc.

Table 11–1. Number of Actions Allowed per State

(a) Address-only state mode

BP/Event	Trace	Jump	Point Timer	Range Timer
4				
3	1			
3		1		
3			1	
2	2			
2		2		
2				1
2			2	
2	1	1		
2	1		1	
2		1	1	
1	3			
1	2	1		
1	2		1	
1	1	2		
1	1			1
1		2	1	
1		1		1
1		1	2	
1			1	1
	4			
	2			1
				2

(b) Address-and-data state mode

BP/Event	Trace	Jump	Point Timer	Range Timer
2				
1	1			
1		1		
1			1	
	2			
	1		1	
				1
			2	

11.5 Jumping to Another State

One of the BTT actions that you can define is a jump to another state. To define a jump action, choose Jump from the Select action menu. (To display the Select action menu, click on <Add action...> in the BTT Setup dialog box.) In the action dialog box for the jump, you can define the conditions for qualifying the jump; you can also define which state the debugger should jump to when the jump is qualified.

This is useful when you need a special method for handling an exception. For example, suppose that you are running your program and collecting traces. During the trace, your program calls an error routine. You may want to trace the error routine differently, then return to the other trace when the error routine completes. To do this, you could:

1) Set up one state to collect the main trace samples.

2) If the error routine is called, jump to a second state.

3) While the error routine is executing, collect trace samples related to the routine.

4) When the error routine completes, jump back to the previous state and continue collecting trace samples under the original conditions.

11.6 Using Hardware Breakpoints and Events

To define a BP/event, choose BP/event from the Select action menu. (To display the Select action menu, click on <Add action...> in the BTT Setup dialog box.) This enables you to define a set of conditions that can cause a hardware breakpoint. A hardware breakpoint halts both the CPU and the BTT.

Note:

In many cases, you will want a breakpoint to occur as soon as the BP/event conditions are met. In this case, once you have defined the conditions, you are finished—it is not necessary for you to read the remainder of this section.

In some cases, you may want to do something more complex—for example, you may want the BP/event to occur several times before the breakpoint occurs, or you may want to collect a certain number of traces after the BP/event is detected. For these cases, you can use the following components, which interact to define the point at which a hardware breakpoint occurs:

❏ The **event counter** counts the number of times BP/event conditions are met for a state. The default value of the event counter is 1; its maximum value is 0FFFFh ($65,535_{10}$). You can define the event counter value in this part of the BTT Setup dialog box:

```
Event count [1 ...... ]
```

Each state has its own event counter. If a state defines more than one BP/event, the event counter counts the number of times any of the state's BP/event conditions are met.

Note that if you set the event counter to 0, you disable the BP/event feature for that state.

❏ The **end state** defines the last state in a sequence of states. The default end state is state 0. The end state is a global setting.

❏ The **loop counter** defines the number of times the BTT should sequence through the states before a breakpoint can occur. The default value of the loop counter is 1; its maximum value is 0FFFFh ($65,535_{10}$). The loop counter is a global setting.

❏ The **delay counter** defines the number of trace samples that are taken after a breakpoint occurs. (The CPU and BTT are not halted by the breakpoint until the trace samples are collected.) The default value of the delay counter is 0; its maximum value is 07FFh (2047_{10}). The delay counter is a global setting.

To access global settings, click on <Globals> in the BTT Setup dialog box; you'll see the Globals dialog box shown in Figure 11–5.

Figure 11–5. The Global Settings Dialog Box

```
 Globals 
Delay count   [0 ................ ]
Max trace     [0 ................ ]
End state     [0 ................ ]
Loop count    [1 ................ ]
Time out      [0000.000 000 000]

              <<OK>>    <Cancel>
```
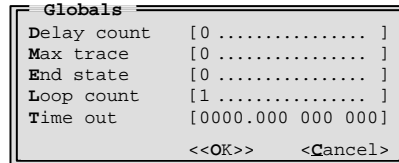
Figure 11–6 shows how the event counter, end state, loop counter, and delay counter can interact. Figure 11–6 shows three levels of complexity; a hardware breakpoint could occur at any of these three levels, depending on what you want to accomplish. You can set up complex state sequencing and trace collection, or you can set up a simple hardware breakpoint.

Figure 11–6. Sequence of Events That Determine When a Breakpoint Can Occur

1) Each time the BP/event conditions are met, the event counter is decremented.

2) When the event counter reaches 0, the BTT moves to the next sequential state. This continues for each state in the sequence until the end state is reached.

3) When the end state's event counter reaches 0, the loop counter is decremented.

4) The BTT then goes back to state 0 and continues sequencing through the states in this manner until the loop counter reaches 0.

5) When the loop counter reaches 0, the delay counter is decremented each time trace conditions are met (as defined for the end state).

6) When the delay counter reaches 0, the hardware breakpoint (defined by the end state) occurs.

Basic breakpointing

If you want a hardware breakpoint to occur without having to sequence through several states, follow these steps:

1) Make sure you are in state 0.

2) Define the BP/event conditions.

3) If you want the hardware breakpoint to occur as soon as the BP/event conditions are met, leave state 0's event counter in the default setting (1). If you would like the breakpoint to occur after the conditions have been met several times, set the event counter to the number of times you want the event to occur before breakpointing.

4) Leave the end state, the loop counter, and the delay counter at their default settings (end state=state 0, loop counter=1, delay counter=0).

Because the current state and the end state are the same, and the loop and delay counters are at their default settings, no decrementing or sequencing occurs. This allows the breakpoint to occur immediately (or as soon as a certain number of events occur, if you're using the event counter).

Sequencing before a breakpoint

Sometimes it's useful to repeat the actions performed in a series of states. For example, this would be a good way to use the point or range timer to collect the average time for a situation such as subroutine execution. To sequence through states, follow these steps:

1) Decide how many times you want to sequence through states; assign this value to the loop counter.

2) Select an end state (the beginning state for any sequence is always state 0; you can use any of the four states as the end state).

3) If you are using the point or range timer, define the starting and stopping conditions for them in the appropriate state.

4) Define the BP/event conditions in the selected end state.

5) Leave the delay counter at its default setting.

Collecting traces, then breakpointing

You can use the delay counter to collect traces in combination with a hardware breakpoint. When the BP/event conditions are met for the end state, the BTT will collect a number of traces before taking the breakpoint. The end state must define conditions for a trace action; the delay counter defines how many trace samples will be collected before the breakpoint is taken.

This is useful for determining what happens following a particular event. You can define the BP/event for the event in question, then gather trace samples; after the trace samples are collected and the breakpoint is taken, you can examine the trace buffer.

11.7 Collecting Trace Samples

A trace sample is a snapshot of the processor status, taken synchronously with the '370 clock. This snapshot shows the information on the address and data buses, cycle information, timing information, and the reverse assembly of associated code (when appropriate).

To collect trace samples, choose Trace from the Select action menu. (To display the Select action menu, click on <Add action...> in the BTT Setup dialog box.) The debugger will display an action dialog box where you can define a set of conditions that will cause bus cycle information to be stored in the trace buffer.

The trace buffer is a circular buffer that can hold up to 2047 trace samples, numbered 0–2046. By default, if more than 2047 trace samples meet the trace conditions, they will continue to be written to the trace buffer, overwriting existing samples beginning with sample 0.

You can change this behavior by using a max trace value. Max trace is a global setting. (To access the global settings, click on <Globals> in the BTT Setup dialog box.) The default value for max trace is 0, which enables the overwrite behavior. If you choose a nonzero max-trace value, the BTT will halt after it collects max trace number of samples. This prevents overwriting of the trace buffer. The maximum value for max trace is 2047.

Trace modes

In addition to defining whether a memory read, memory write, or instruction acquisition cycle can be collected as a trace sample, you can use the trace mode to define whether cycles that are *associated with* a qualified trace sample can also be collected. There are two trace modes, normal mode and TRIX (trace instruction extended) mode, which are defined in this part of the BTT Setup dialog box:

```
┌ Trace Mode ┐
(*) Normal
( ) TRIX
```

The trace mode defines which cycles can be stored in the trace buffer:

❑ **Qualified cycles only.** When you select **normal mode**, only trace samples that meet the conditions you've defined will be stored in the trace buffer. Normal mode is the default trace mode.

❑ **Qualified IAQ cycles plus associated reads and writes.** When you select **TRIX mode**, if an instruction opcode qualifies as a trace sample, then all reads and writes associated with that instruction are also stored in the trace buffer. (Note that the first byte of an instruction must qualify, or no associated cycles can qualify.)

11.8 Using the BTT Timers

The BTT has three types of timers:

❏ The **point timer** is an action that can be used for collecting timing statistics.

❏ The **range timer** is an action that can be used for collecting timing statistics.

❏ The **time-out timer** is a global setting that limits the amount of time that your program can run.

Collecting timing statistics

The point timer or range timer can be defined by selecting Point Timer or Range Timer from the Select action menu. (To display the Select action menu, click on <Add action...> in the BTT Setup dialog box.)

The point and range timers operate similarly, tracking the amount of time that elapses from the point at which one condition occurs to the point at which a second condition occurs. The difference between these timers involves the types of conditions that can cause the timers to start and stop:

❏ The **point timer** starts and stops on address values. Other options—data values, memory cycle types, etc.—can also be used to qualify starting and stopping, but they are shared; the only condition that varies is the address.

When you select Point Timer from the Select action menu, the debugger displays a dialog box where you can define the conditions for starting and stopping the timer. You must define two address values; the addr1 field defines the starting address, and the addr2 field defines the stopping address.

❏ The **range timer** starts and stops on independent sets of conditions.

When you select Range Timer from the Select action menu, the debugger displays a dialog box where you can define the conditions for starting the range timer; when you finish defining the conditions and click on <OK>, the debugger displays a second dialog box where you can define the conditions for stopping the range timer. Because you must select separate starting and stopping conditions, the range timer counts as two actions for a state.

You can define a total of two timer actions per state—one point timer and one range timer action, two point timer actions, or two range timer actions.

The timers track in a cumulative manner; they can be started and stopped many times. When a timer is started for the first time, it begins counting from 0. If a timer restarts, instead of starting from 0, it adds to the previous count so that it tracks the total time for the BTT session.

Timing information is reported along with trace information in the INSPECT window. The information is listed for timers that are labeled as *timer 1* and *timer 2*. The numbers refer to which timer action you defined first; the INSPECT window will report the first timer action you defined for a state as timer 1 statistics and the second timer action that you defined for the same state as timer 2 statistics. For example, if you defined two point timer actions in state 0, the first one would be reported as timer 1, and the second would be reported as timer 2. The action dialog box will tell you if you're defining timer 1 or timer 2 by displaying the phrase *Timer #1* or *Timer #2* in the upper right corner of the box.

If you move to another state and define more timer actions, again, the first timer you define for the state will be reported as timer 1 and the second as timer 2. These statistics are not reported independently—they are combined with the timer statistics that are already being collected for timer 1 and timer 2. The timing statistics shown in the INSPECT window will be accumulated statistics for timer 1 and timer 2. Unless you are dealing with a special case, you may want to restrict your timer definitions to a single state.

For more information, see *Interpreting point and range timer statistics* on page 11-24.

Limiting program run time

You can use the time-out timer to define a time limit for running your program.

The time-out timer is one of the global settings. (To access the global settings, click on <Globals> in the BTT Setup dialog box.) The time-out value is defined in seconds, milliseconds, microseconds, and nanoseconds:

```
0000.000   000   000
    │        │     └─ nanoseconds
    │        └─ microseconds
    └─ milliseconds
  └─ seconds
```
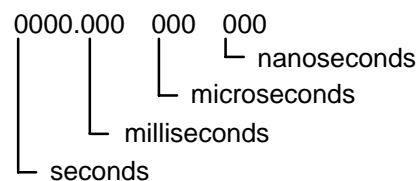
The default setting is 0, which allows your program to run without a time limit. Any setting other than 0 will cause your program to halt after it has run for the specified time.

11.9 Viewing Trace Buffer and Timing Information

To view the trace buffer, enter the INSP command or select Inspect from the BTT pulldown menu. The debugger will open the INSPECT window, which looks like the window shown in Figure 11–7.

Figure 11–7. An Example of the INSPECT Window
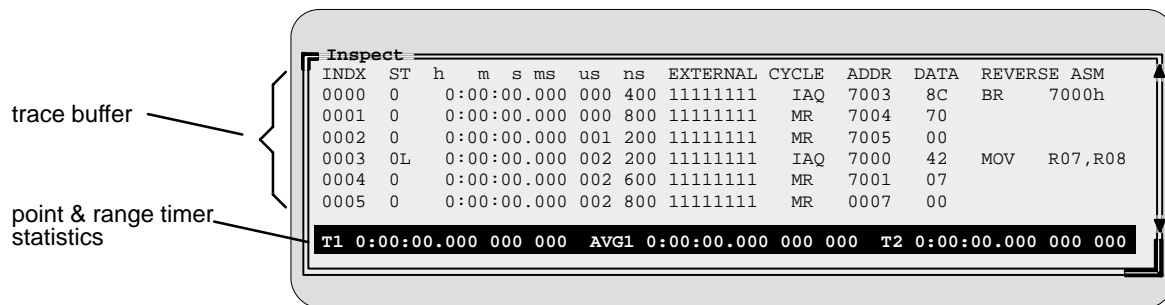
The INSPECT window shows trace buffer information as well as point and range timer statistics.

Interpreting trace buffer information

The trace-buffer portion of the INSPECT window shows the following information:

❑ **INDX field.** This field shows the trace sample's number in the buffer. (The trace buffer can hold up to 2047 samples.)

❑ **ST field.** This field shows which state the BTT was in when the sample was collected. Additionally, next to the state number, you will see an **E** if the trace sample also met BP/event conditions or an **L** if the trace sample was the last event and caused a breakpoint to occur.

❑ **h to ns fields.** The next several fields show timing information about the trace sample relative to the previous sample. The timing information is formatted in hours, minutes, seconds, microseconds, and nanoseconds.

By default, the time shown is the total amount of time that has passed from the start of tracing to the time a trace sample was collected. However, the INSPECT window can report the time in three ways. To select another method, choose Format from the BTT menu. You'll see a dialog box like the one in Figure 11–8.

Figure 11–8. Selecting the Trace Sample Timing Format

```
┌─ Format Time Stamp ═══════════════════╗
│ (*)Absolute  ( )Delta    ( )Mark      │
│                                       │
│                      <<OK>>  <Cancel> │
└───────────────────────────────────────┘
```
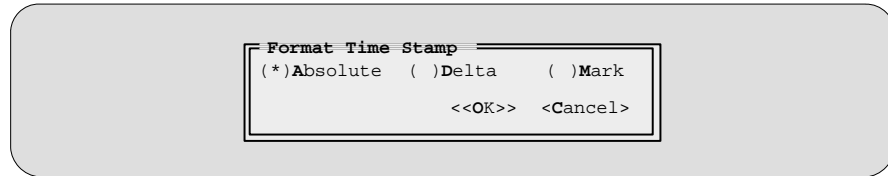
Your choices for formatting the timing information are:

■ **Total time.** The **Absolute** setting is the default, showing the total amount of time that has passed from the start of tracing to the time the trace sample was collected.

■ **Difference between samples.** Select the **Delta** setting to show the time difference between any trace sample and the sample that precedes it.

■ **Difference from a specific sample.** Select the **Mark** setting to show the time difference between any trace sample and a specific sample within the trace buffer; the specific sample is selected by the position of the cursor within the INSPECT window. (For methods of locating a particular sample, see *Viewing selected trace samples* on page 11-22.) Samples collected before the selected sample show a negative time difference; samples collected after the selected sample show a positive time difference.

❑ **EXTERNAL field.** This field shows the values on the external probes at the time the trace sample was collected. A 1 indicates a high signal, and a 0 indicates a low signal.

❑ **CYCLE field.** This field shows what type of memory cycle took place when the trace sample was collected. You will see one of three codes in this field:

    MR   The trace sample was collected during a memory-read cycle.

    MW   The trace sample was collected during a memory-write cycle.

    IAQ   The trace sample was collected during an instruction-acquisition cycle.

❑ **ADDR field.** This field shows the value that was on the address bus when the trace sample was collected.

❑ **DATA field.** This field shows the value that was on the data bus when the trace sample was collected.

❑ **REVERSE ASM field.** This field shows the assembly language instruction associated with the trace sample.

Viewing selected trace samples

There are several ways to move around in the INSPECT window. You can scroll through the INSPECT window just as you can scroll through any other window. You can also look for a specific trace sample, either by its position within the trace buffer or by its conditions.

To look for a trace sample by its position, select Position from the BTT menu; you'll see a dialog box like the one in Figure 11–9. Fill in the index number of a specific trace sample, or click on <Bottom> to go to the end of the trace buffer.

Figure 11–9. Locating a Trace Sample by Its Index Number

```
┌─ BTT Position ─────────────────────────────────────────────────┐
│ Sample Number [0 .................... ]            <Bottom>     │
│                                          <<OK>> <Cancel>       │
└────────────────────────────────────────────────────────────────┘
```

To look for a trace sample that meets specific conditions, select Lookup from the BTT menu; you'll see a dialog box like the one in Figure 11–10.

Figure 11–10.Locating a Trace Sample by Its Conditions

```
┌─ Locate ───────────────────────────────────────────────────────────┐
│                    Flag : (*)None  ( )Event  ( )Last               │
│  ┌─ Address qualifiers ─────────────────────────────────────────┐  │
│  │ (*)One point    ( )Two points    ( )In range    ( )Outside range│
│  └──────────────────────────────────────────────────────────────┘  │
│ addr1=[0x0000 ................................................... ] │
│ addr2=[N/A                                                        ] │
│  mask [0x0000 ......... ]                                           │
│  ┌─ Data qualifiers ────────────────────────────────────────────┐  │
│  │ (*)One point    ( )Two points    ( )In range    ( )Outside range│
│  └──────────────────────────────────────────────────────────────┘  │
│ data1=[0x00 ..................................................... ] │
│ data2=[N/A                                                        ] │
│  mask [0x00 ........... ]                                           │
│                                       extern [0x00 .............. ] │
│  ┌─ Cycle ─────────────────────┐        mask [0x00 ....... ]        │
│  │ [X] MR   [X] MW   [X] IAQ   │                                    │
│  └─────────────────────────────┘         <<OK>>    <Cancel>        │
└────────────────────────────────────────────────────────────────────┘
```

Supply information in this dialog box in the same way that you would if you were defining an action; the debugger will look for a trace sample that defines the conditions you supply. Use the Flag field to indicate whether you are searching for a specific type of event. Your choices are:

❑ **No event.** Select None if you want to look for a trace sample that was not an event. None is the default selection.

❑ **Simple event.** Select Event if you want to look for a trace sample that also met BP/event conditions.

❑ **Last event and breakpoint.** Select Last if you want to look for a trace sample that was the last event and that caused a hardware breakpoint to occur.

Storing trace buffer contents to a file

It can be useful to collect several sets of trace samples and compare the results. The easiest way to do this is to save the contents of the trace buffer before you begin to collect more samples. To do this, select Save from the BTT menu. You'll see a dialog box like the one in Figure 11–11.

Figure 11–11. Saving the Trace Buffer

```
┌─Save trace buffer──────────────────────────────────────────────────┐
│ File name [ ..................................................... ]  │
│ Start [0....]  to  End [299...]                                       │
│                                              <<OK>>    <Cancel>       │
└──────────────────────────────────────────────────────────────────────┘
```
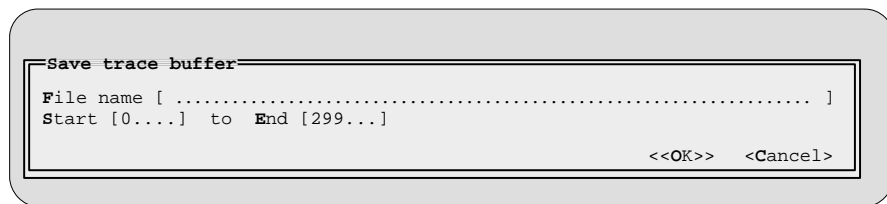
The dialog box asks you for the name of the file where you'd like to store the trace buffer contents. The dialog box also asks you which trace samples should be stored; the Start field specifies the first sample, and the End field specifies the last. By default, all samples are stored.

When you examine the file, the information will be in the same format in which it was displayed in the INSPECT window.

You can also use the TSAVE command to save the contents of the trace buffer. The syntax for this command is:

tsave *filename*

where *filename* names the file that contains the saved information.

Interpreting point and range timer statistics

The INSPECT window shows statistics for two timers, labeled timer 1 and timer 2. These timers correspond to point or range timer actions. The timer action you defined first for a state is represented as timer 1, and the timer action you defined second for the same state is represented as timer 2. For more information about point and range timer actions, see *Collecting timing statistics* on page 11-18.

The INSPECT window shows the total times for both timers. It also shows the average for timer 1. (The average is the total time divided by the number of times the timer was started.)

11.10 Reusing a BTT Setup

You can save the current BTT setup—all of the global settings and all of the actions you've defined for each of the states—so that you can use the setup again. To do this, click on <Store> in the BTT Setup dialog box. You'll see a dialog box like the one in Figure 11–12. This dialog box asks for the name of the file where you'd like to save the information.

Figure 11–12. Saving the Current BTT Setup

```
┌ Store BTT ══════════════════════════════════════════════════════════════
│ File name [ ..................................................................... ]
│                                                          <<OK>>   <Cancel>
```
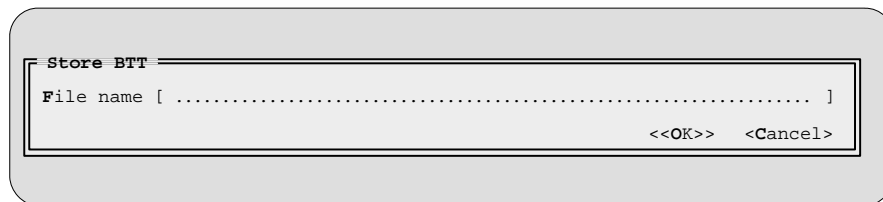
When you want to reuse a saved BTT setup, open the BTT Setup dialog box and click on <Load>. You'll see a dialog box like the one in Figure 11–13. This dialog box asks for for the name of the file that contains the saved setup.

Figure 11–13. Loading a Saved BTT Setup

```
┌ Load BTT ═══════════════════════════════════════════════════════════════
│ File name [ ..................................................................... ]
│                                                          <<OK>>   <Cancel>
```
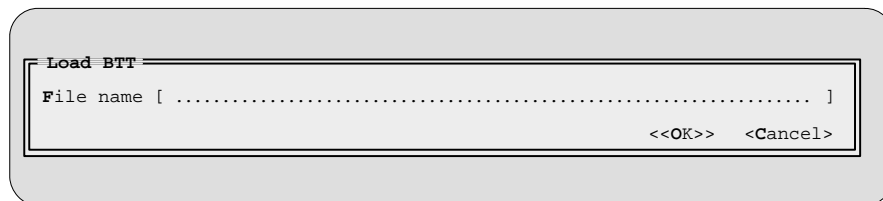
You can also use the BTT command to load a saved setup. The syntax for this command is:

BTT *filename*

where *filename* names the file that contains the saved setup.

Profiling Code Execution

The profiling environment is a special debugger environment in which you can collect execution statistics for your code. Only the XDS/22 emulation system using MicroSoft Windows supports the profiling environment.The profiling environment is *separate* from the basic debugging environment; the only way to switch between the two environments is by exiting and then reinvoking the debugger.

This chapter describes the general profiling process as well as the differences between the basic debugging environment and the profiling environment.

12.1 An Overview of the Profiling Process

Profiling consists of five simple steps:

Step 1

Enter the profiling environment. See *Entering the Profiling Environment*, page 12-4.

Step 2

Identify the areas of code where you'd like to collect statistics. See *Defining Areas for Profiling*, page 12-6.

Step 3

Identify the profiling session stopping points. See *Defining a Stopping Point,* page 12-14.

Step 4

Begin profiling. See *Running a Profiling Session,* page 12-16.

Step 5

View the profile data. See *Viewing Profile Data,* page 12-18.

Note:

When you compile a program that will be profiled, you must use the –g and the –as options. The –g option includes symbolic debugging information; the –as option ensures that you will be able to include ranges as profile areas.

A profiling strategy

The profiling environment provides a method for collecting execution statistics about specific areas in your code. This gives you immediate feedback on your application's performance.Here's a suggestion for a basic approach to optimizing the performance of your program.

1) Mark all the functions in your program as profile areas.

2) Run a profiling session; find the busiest functions.

3) Unmark all the functions.

4) Mark the individual lines in the busy functions and run another profiling session.

12.2 Entering the Profiling Environment

To enter the profiling environment, invoke the debugger with the **–profile** option. At the system command line, enter:

xds370w –profile ⏎

Use any additional debugger options that you desire (–b, –p, etc.).

Restrictions of the profiling environment

In addition to the special features supported by the profiling environment, several restrictions apply to the profiling environment:

❑ You'll always be in mixed mode.

❑ COMMAND, DISASSEMBLY, FILE, and PROFILE are the only windows available; additional windows, such as the WATCH window, cannot be opened.

❑ Breakpoints cannot be set. (However, you can use a similar feature called *stopping points* in marking sections of code for profiling.)

❑ The profiling environment supports only a subset of the debugger commands. Table 12–1 lists the debugger commands that can and can't be used in the profiling environment.

Table 12–1.Debugger Commands That Can/Can't be Used in the Profiling Environment

Can be used		Can't be used	
?	MOVE	ADDR	MS
ALIAS	MR	ASM	NEXT
CD	PROMPT	BA	PATCH
CLS	QUIT	BD	RETURN
DASM	RELOAD	BL	RRUNF
DIR	RESET	BORDER	RUN
DLOG	RESTART	BR	RUNB
ECHO	RRUN	BTT	RUNF
EVAL	SCONFIG	C	SCOLOR
FILE	SIZE	CALLS	SETF
FUNC	SLOAD	CNEXT	SOUND
IF/ELSE/	SYSTEM	COLOR	SSAVE
ENDIF	TAKE	CSTEP	STEP
LOAD	UNALIAS	DISP	TSAVE
LOOP/	USE	FILL	WA
ENDLOOP	VERSION	GO	WD
MA	WIN	HALT	WHATIS
MAP	WRUN	INSP	WR
MD	ZOOM	MEM	WRUNF
ML		MIX	

Be sure you don't use any of the "can't be used" commands in your initialization batch file.

Using pulldown menus in the profiling environment

The debugger displays a different menu bar in the profiling environment:

```
Load  mAp  Mark  Enable  Disable  Unmark  View  Stop-points  Profile
```

The Load and mAp menus correspond to the Load and Map menus available in the basic debugger environment. The other entries provide access to profiling commands and features.

The profiling environment's pulldown menus operate similarly to the basic debugger pulldown menus. However, several of the menus have additional submenus. A submenu is indicated by a > character following a menu item. For example, here's one of the submenus for the Mark menu:

```
       Mark
  C level        >
  Asm level      >    Line areas       >
                      Range areas      >    Explicitly
                      Function areas >      in one Function
```
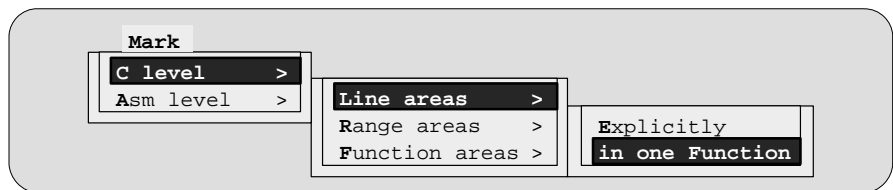
Chapter 5, *Entering and Using Commands*, shows which debugger commands are associated with the menu items in the basic debugger pulldown menus. Because the profiling environment supports over 100 profile-specific commands, it's not practical to show the commands associated with the menu choices. Here's a tip to help you with the profiling commands: the highlighted menu letters form the name of the corresponding debugger command. For example, if you prefer the function-key approach to using menus, the highlighted letters in **M**ark→ **C** level→**L**ine areas→in one **F**unction show that you could press ALT M, C, L, F. This also shows that the corresponding debugger command is MCLF.

12.3 Defining Areas for Profiling

Within the profiling environment, you can collect statistics on three types of areas:

❑ **Individual lines** in C or disassembly
❑ **Ranges** in C or disassembly
❑ **Functions** in C only

To identify any of these areas for profiling, you *mark* the line, range, or function. You can disable areas so they won't affect the profile data. You can re-enable areas that have been disabled. And, you can unmark areas that you are no longer interested in.

The mouse provides a means of accomplishing the simplest marking, disabling, enabling, and unmarking tasks. The pulldown menus also support these tasks; additionally, they provide a means of accomplishing more complex tasks.

The following subsections explain how to mark, disable, re-enable, and unmark profile areas by using the mouse or the pulldown menus. The individual commands are summarized in *Restrictions of the profiling environment* on page 12-4.

Marking an area

Marking an area qualifies it for profiling so that the debugger can collect timing statistics about the area.

Below are directions for using the mouse to mark a line area, a range area, or a function area. Remember, to display C code, use the FILE or FUNC command; to display disassembly, use the DASM command.

Note:

❑ Marking an area in C *does not* mark the associated code in disassembly.

❑ Areas can be nested; for example, you can mark a line within a marked range. The debugger will report statistics for both the line and the function.

❑ Ranges cannot overlap and they cannot span function boundaries.

Marking a line. These instructions apply to both C and disassembly.

1) Point to the line you want to mark.

2) Click the left mouse button.

   The beginning of the line will be highlighted with a blinking >>*.*

3) Click the left mouse button again.

   The beginning of the line will be highlighted with Le> *(line enabled).*

Marking a range. These instructions apply to both C and disassembly.

1) Point to the first line of the range you want to mark.

2) Click the left mouse button.

   The beginning of the line will be highlighted with a blinking >>*.*

3) Point to the last line of the range.

4) Click the left mouse button again.

   The beginning of the line will be highlighted with Re> *(range enabled), marking the beginning of the range. The last line will be highlighted with* <<*, marking the end of the range.*

Marking a function. These instructions apply to C only.

1) Point to the marked line.

2) Click the left mouse button.

   The beginning of the line will be highlighted with Fe> *(function enabled).*

key

Table 12–2 lists the menu selections for marking areas. The highlighted areas show the keys that you can use if you prefer to use the function-key method of selecting menu choices.

Table 12–2. Menu Selections for Marking Areas

To mark this area	C only: **M**ark→**C** level	Disassembly only: **M**ark→**A**sm level
Lines	→**L**ine areas	→**L**ine areas
▢ By line number†	→**E**xplicitly	→**E**xplicitly
▢ All lines in a function	→in one **F**unction	→in one **F**unction
Ranges	→**R**ange areas	→**R**ange areas
▢ By line numbers†	→**E**xplicitly	→**E**xplicitly
Functions	→**F**unction areas	
▢ By function name	→**E**xplicitly	
▢ All functions in a module	→in one **M**odule	not applicable
▢ All functions everywhere	→**G**lobally	

† C areas are identified by line number; disassembly areas are identified by address.

Disabling an area

At times, it is useful to identify areas that must not impact profile statistics. To do this, you should *disable* the appropriate area. Disabling effectively subtracts the timing information of the disabled area from all profile areas that include or call the disabled area. Areas must be marked before they can be disabled.

For example, if you have marked a function that calls a standard C function such as malloc(), you may not want malloc() to affect the statistics for the calling function. You could mark the line that calls malloc(), and then disable the line. This way, the profile statistics for the function would not include the statistics for malloc().

Note:

If you disable an area after you've already collected statistics on it, that information will be lost.

The simplest way to disable an area is to use the mouse, as described below.

Disabling a line area:

1) Point to the marked line.

2) Click the left mouse button once.

   The beginning of the line will be highlighted with Ld> *(line disabled).*

Disabling a range area:

1) Point to the marked line.

2) Click the left mouse button once.

   The beginning of the line will be highlighted with Rd> *(range disabled).*

Disabling a function area:

1) Point to the marked statement that defines the function.

2) Click the left mouse button once.

   The beginning of the line will be highlighted with Fd> *(function disabled).*

key

Table 12–3 lists the menu selections for disabling areas. The highlighted areas show the keys that you can use if you prefer to use the function-key method of selecting menu choices.

Table 12–3. Menu Selections for Disabling Areas

To disable this area	C only: Disable→C level	Disassembly only: Disable→Asm level	C *and* disassembly: Disable→Both levels
Lines	→**L**ine areas	→**L**ine areas	→**L**ine areas
By line number†	→**E**xplicitly	→**E**xplicitly	not applicable
All lines in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
All lines in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
All lines everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Ranges	→**R**ange areas	→**R**ange areas	→**R**ange areas
By line numbers†	→**E**xplicitly	→**E**xplicitly	not applicable
All ranges in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
All ranges in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
All ranges everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Functions	→**F**unction areas		→**F**unction areas
By function name	→**E**xplicitly		not applicable
All functions in a module	→in one **M**odule	not applicable	→in one **M**odule
All functions everywhere	→**G**lobally		→**G**lobally
All areas	→**A**ll areas	→**A**ll areas	→**A**ll areas
All areas in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
All areas in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
All areas everywhere	→**G**lobally	→**G**lobally	→**G**lobally

† C areas are identified by line number; disassembly areas are identified by address.

Re-enabling a disabled area

When an area has been disabled and you would like to profile it once again, you must enable the area. To use the mouse, just point to the line, the function, or the first line of a range, and click the left mouse button; the range will once again be highlighted in the same way as a marked area.

key

In addition to using the mouse, the debugger supports an entire set of commands for enabling areas. These commands are easiest to enter by using the Enable menu. Table 12–4 lists these menu selections.

Table 12–4.Menu Selections for Enabling Areas

To enable this area	C only: **E**nable→**C** level	Disassembly only: **E**nable→**A**sm level	C *and* disassembly: **E**nable→**B**oth levels
Lines	→**L**ine areas	→**L**ine areas	→**L**ine areas
By line number†	→**E**xplicitly	→**E**xplicitly	not applicable
All lines in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
All lines in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
All lines everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Ranges	→**R**ange areas	→**R**ange areas	→**R**ange areas
By line numbers†	→**E**xplicitly	→**E**xplicitly	not applicable
All ranges in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
All ranges in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
All ranges everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Functions	→**F**unction areas		→**F**unction areas
By function name	→**E**xplicitly		not applicable
All functions in a module	→in one **M**odule	not applicable	→in one **M**odule
All functions everywhere	→**G**lobally		→**G**lobally
All areas	→**A**ll areas	→**A**ll areas	→**A**ll areas
All areas in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
All areas in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
All areas everywhere	→**G**lobally	→**G**lobally	→**G**lobally

† C areas are identified by line number; disassembly areas are identified by address.

Unmarking an area

If you want to stop collecting information about a specific area, unmark it. You can use the mouse (described below.

Unmarking a line area:

1) Point to the marked line.

2) Click the right mouse button once.

The line will no longer be highlighted.

Unmarking a range area:

1) Point to the marked line.

2) Click the right mouse button once.

The line will no longer be highlighted.

Unmarking a function area:

1) Point to the marked statement that defines the function.

2) Click the right mouse button once.

The line will no longer be highlighted.

Table 12–5 lists the selections on the Unmark menu.

Table 12–5.Menu Selections for Unmarking Areas

To unmark this area	C only: **U**nmark→**C** level	Disassembly only: **U**nmark→**A**sm level	C *and* disassembly: **U**nmark→**B**oth levels
Lines By line number† All lines in a function All lines in a module All lines everywhere	→**L**ine areas →**E**xplicitly →in one **F**unction →in one **M**odule →**G**lobally	→**L**ine areas →**E**xplicitly →in one **F**unction →in one **M**odule →**G**lobally	→**L**ine areas not applicable →in one **F**unction →in one **M**odule →**G**lobally
Ranges By line numbers† All ranges in a function All ranges in a module All ranges everywhere	→**R**ange areas →**E**xplicitly →in one **F**unction →in one **M**odule →**G**lobally	→**R**ange areas →**E**xplicitly →in one **F**unction →in one **M**odule →**G**lobally	→**R**ange areas not applicable →in one **F**unction →in one **M**odule →**G**lobally
Functions By function name All functions in a module All functions everywhere	→**F**unction areas →**E**xplicitly →in one **M**odule →**G**lobally	not applicable	→**F**unction areas not applicable →in one **M**odule →**G**lobally
All areas All areas in a function All areas in a module All areas everywhere	→**A**ll areas →in one **F**unction →in one **M**odule →**G**lobally	→**A**ll areas →in one **F**unction →in one **M**odule →**G**lobally	→**A**ll areas →in one **F**unction →in one **M**odule →**G**lobally

† C areas are identified by line number; disassembly areas are identified by address.

12.4 Defining a Stopping Point

Before you run a profiling session, you must identify the point where the debugger should stop collecting statistics. By default, C programs contain an *exit* label, and this is defined as the default stopping point when you load your program. (You can delete exit as a stopping point, if you wish.) If your program does not contain an exit label, or if you prefer to stop at a different point, you can define another stopping point. You can set multiple stopping points; the debugger will stop at the first one it finds.

Each stopping point is highlighted in the FILE or DISASSEMBLY window with a ***** character at the beginning of the line. Even though no statistics can be gathered for areas following a stopping point, the areas will be listed in the PROFILE window.

You can use the mouse or commands to add or delete a stopping point; you can also use commands to list or reset all the stopping points.

Note:

You cannot set a stopping point on a statement that has already been defined as a part of a profile area.

To set a stopping point:

1) Point to the statement that you want to add as a stopping point.

2) Click the right mouse button.

To remove a stopping point:

1) Point to the statement marking the stopping point that you want to delete.

2) Click the right mouse button.

The debugger supports several commands for adding, deleting, resetting, and listing stopping points (described below); all of these commands can also be entered from the Stop-points menu.

sa   To add a stopping point, use the SA (stop add) command. The syntax for this command is:

    sa *address*

This adds *address* as a stopping point. The *address* parameter can be a label, a function name, or a memory address.

sd   To delete a stopping point, use the SD (stop delete) command. The syntax for this command is:

    sd *address*

This deletes *address* as a stopping point. As for SA, the *address* can be a label, a function name, or a memory address.

sr   To delete all the stopping points at once, use the SR (stop reset) command. The syntax for this command is:

    sr

This deletes all stopping points, including the default *exit* (if it exists).

sl   To see a list of all the stopping points that are currently set, use the SL (stop list) command. The syntax for this command is:

    sl

12.5 Running a Profiling Session

Once you have defined profile areas and a stopping point, you can run a profiling session. You can run two types of profiling sessions:

❑ A **full profile** collects a full set of statistics for the defined profile areas.

❑ A **quick profile** collects a subset of the available statistics (it doesn't collect exclusive or exclusive max data, which are described in Section 12.6). This reduces overhead because the debugger doesn't have to track entering/exiting subroutines within an area.

The debugger supports commands for running both types of sessions. In addition, the debugger supports a command that helps you to resume a profiling session. All of these commands can also be entered from the Profile menu.

pf   To run a full profiling session, use the PF (profile full) command. The syntax for this command is:

pf   *starting point* [*, update rate*]

pq   To run a quick profiling session, use the PQ (profile quick) command. The syntax for this command is:

pq   *starting point* [*, update rate*]

The debugger will collect statistics on the defined areas between the *starting point* and the stopping point. The *starting point* parameter can be a label, a function name, or a memory address. There is no default starting point.

The *update rate* is an optional parameter that determines how often the statistics listed in the PROFILE window will be updated. The *update rate* parameter can have one of these values:

0   An *update rate* of 0 means that the statistics listed in the PROFILE window are not updated until the profiling session is halted. A "spinning wheel" character will be shown at the beginning of the PROFILE window label line to indicate that a profiling session is in progress. 0 is the default value.

≥1   If a number greater than or equal to 1 is supplied, the statistics in the PROFILE window are updated during the profiling session. If a value of **1** is supplied, the data will be updated as often as possible. When larger numbers are supplied, the data is updated less often.

<0   If a negative number is supplied, the statistics listed in the PROFILE window are not updated until the profiling session is halted. The "spinning wheel" character is not displayed.

No matter which *update rate* you choose, you can force the PROFILE window to be updated during a profiling session by pointing to the window header and clicking a mouse button.

After you enter a PF or PQ command, your program is restarted and run up to the defined starting point. Profiling begins when the starting point is reached and continues until a stopping point is reached or until you halt the profiling session by pressing ESC.

pr Use the PR command to resume a profiling session that has halted. The syntax for this command is:

pr   [*clear data*   [, *update rate*]]

The optional *clear data* parameter tells the debugger whether or not it should clear out the previously collected data. The *clear data* parameter can have one of these values:

0 The profiler will continue to collect data, adding it to the existing data for the profiled areas, and to use the previous internal profile stacks. 0 is the default value.

nonzero All previously collected profile data and internal profile stacks are cleared.

The *update rate* parameter is the same as for the PF and PQ commands.

12.6 Viewing Profile Data

The statistics collected during a profiling session are displayed in the PRO-FILE window. Figure 12–1 shows an example of this window.

Figure 12–1. An Example of the PROFILE Window

The example in Figure 12–1 shows the PROFILE window with some default conditions:

❑ Column headings show the labels for the default set of profile data, including *Count, Inclusive, Incl-Max, Exclusive*, and *Excl-Max*.

❑ The data is sorted on the address of the first line in each area.

❑ All marked areas are listed, including disabled areas.

You can modify the PROFILE window to display selected profile areas or different data; you can also sort the data differently. The following subsections explain how to do these things.

Note:

To reset the PROFILE display back to its default characteristics, use View→Reset.

Viewing different profile data

By default, the PROFILE window shows a set of statistics labelled as Count, Inclusive, Incl-Max, Exclusive, and Excl-Max. Another field that is not included as part of the default statistics, Address, can also be displayed. Table 12–6 describes the statistic that each field represents.

Table 12–6. Types of Data Shown in the PROFILE Window

Label	Profile data
Count	The number of times a profile area is entered during a session.
Inclusive	The total execution time (cycle count) of a profile area, including the execution time of any subroutines called from within the profile area.
Incl-Max (inclusive maximum)	The maximum inclusive time for one iteration of a profile area.
	If the profiled code contains no flow control (such as conditional processing), inclusive-maximum will equal the inclusive timing divided by the count.
Exclusive	The total execution time (cycle count) of a profile area, excluding the execution time of any subroutines called from within the profile area.
	In general, the exclusive data provides the best statistics for comparing the execution time of one profile area to another area.
Excl-Max (exclusive maximum)	The maximum exclusive time for one iteration of a profile area.
Address	The memory address of the line. If the area is a function or range, the Address field shows the memory address of the first line in the area.

In addition to viewing this data in the default manner, you can view each of these statistics individually. The benefit of viewing them individually is that in addition to a cycle count, you are also supplied with a percentage indication and a histogram.

In order to view the fields individually, you can use the mouse—just point to the header line in the PROFILE window and click a mouse button. You can also use the View→Data menu to select the field you'd like to display. When you use the left mouse button to click on the header, fields are displayed individually in the order listed below on the left. (Use the right mouse button to go in the opposite direction.) On the right are the corresponding menu selections.

Count	**V**iew→**D**ata	→**C**ount
Inclusive		→**I**nclusive
Incl-max		→**In**clusive Max
Exclusive		→**E**xclusive
Excl-max		→**Ex**clusive Max
Address		→**A**ddress
Default		→A**l**l
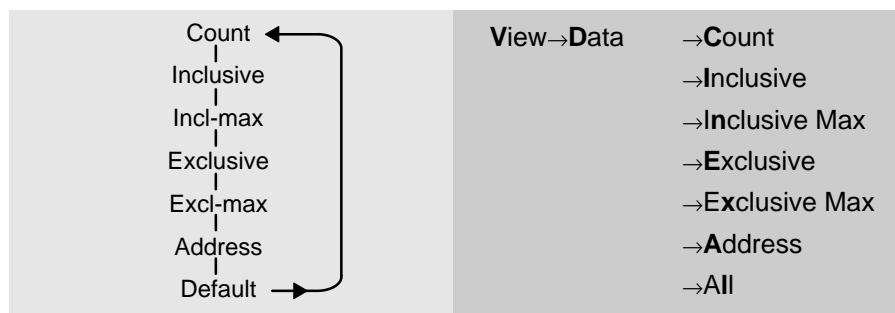
One advantage of using the mouse is that you can change the display while you're profiling.

Data accuracy

During a profiling session, the debugger sets many internal breakpoints and issues a series of RUNB commands. As a result, the processor is momentarily halted when entering and exiting profiling areas. This stopping and starting can affect the cycle count information so that it varies from session to session. This method of profiling is referred to as *intrusive profiling*.

Treat the data as *relative*, not absolute. The percentages and histograms are relevant only to the cycle count from the starting point to the stopping point—not to overall performance. Even though the cycle counts may change if you profiled the same area twice, the relationship of that area to other profiled areas should not change.

Sorting profile data

By default, the data displayed in the PROFILE window is sorted on the memory addresses of the displayed areas. The area with the least significant address is listed first, followed by the area with the most significant address, etc. When you view fields individually, the data is automatically sorted from highest cycle count to lowest (instead of by address).

You can sort the data on any of the data fields by using the View→Sort menu. For example, to sort all the data based on the values of the Inclusive field, use View→Sort→Inclusive; the areas will be redisplayed with the area with the highest Count field displayed first and the area with the lowest area displayed last. This applies even when you are viewing individual fields.

Viewing different profile areas

By default, all marked areas are listed in the PROFILE window. You can modify the window to display selected areas. To do this, use the selections on the View→Filter pulldown menu; these selections are summarized in Table 12–7.

Table 12–7.Menu Selections for Displaying Areas in the PROFILE Window

To view these areas	C only: **V**iew→**F**ilter→**C** level	Disassembly only: **V**iew→**F**ilter→**A**sm level	C *and* disassembly: **V**iew→**F**ilter→**B**oth levels

Lines
❏ By line number
❏ All lines in a function
❏ All lines in a module
❏ All lines everywhere
	→**L**ine areas
	  →**E**xplicitly
	  →in one **F**unction
	  →in one **M**odule
	  →**G**lobally
		→**L**ine areas
		  →**E**xplicitly
		  →in one **F**unction
		  →in one **M**odule
		  →**G**lobally
			→**L**ine areas
			  not applicable
			  →in one **F**unction
			  →in one **M**odule
			  →**G**lobally

Ranges
❏ By line numbers
❏ All ranges in a function
❏ All ranges in a module
❏ All ranges everywhere
	→**R**ange areas
	  →**E**xplicitly
	  →in one **F**unction
	  →in one **M**odule
	  →**G**lobally
		→**R**ange areas
		  →**E**xplicitly
		  →in one **F**unction
		  →in one **M**odule
		  →**G**lobally
			→**R**ange areas
			  not applicable
			  →in one **F**unction
			  →in one **M**odule
			  →**G**lobally

Functions
❏ By function name
❏ All functions in a module
❏ All functions everywhere
	→**F**unction areas
	  →**E**xplicitly
	  →in one **M**odule
	  →**G**lobally
		not applicable	→**F**unction areas
			  not applicable
			  →in one **M**odule
			  →**G**lobally

All areas
❏ All areas in a function
❏ All areas in a module
❏ All areas everywhere
	→**R**ange areas
	  →in one **F**unction
	  →in one **M**odule
	  →**G**lobally
		→**R**ange areas
		  →in one **F**unction
		  →in one **M**odule
		  →**G**lobally
			→**R**ange areas
			  →in one **F**unction
			  →in one **M**odule
			  →**G**lobally

Interpreting session data

General information about a profiling session is displayed in the COMMAND window display area during and after the session. This information identifies the starting and stopping points. It also lists statistics for three important areas:

❏ **Run cycles** shows the number of execution cycles consumed by the program from the starting point to the stopping point.

❏ **Profile cycles** equals the run cycles minus the cycles consumed by disabled areas.

❏ **Hits** show the number of internal breakpoints encountered during the pro-
filing session.

Viewing code associated with a profile area

You can view the code associated with a displayed profile area. The debugger
will update the display so that the associated C or disassembly statements are
shown in the FILE or DISASSEMBLY windows.

Use the mouse to select the profile area in the PROFILE window and display
the associated code:

1) Point to the appropriate area name in the PROFILE window.

2) Click the right mouse button.

The area name and the associated C or disassembly statement will be
highlighted. To view the code associated with another area, point-and-click
again.

If you are attempting to show disassembly, you may have to make several
attempts because program memory can only be accessed when the target is
not running.

12.7 Saving Profile Data to a File

You may want to run several profiling sessions during a debugging session. Whenever you start a new profiling session,the results of the previous session are lost. However, you can save the results of the current profiling session to a system file.There are two ways that you can do this:

vac   To save the contents of the PROFILE window to a system file, use the VAC (view save current) command. The syntax for this command is:

vac *filename*

This saves only the current view; if, for example, you are viewing only the Count field, then only that information will be saved.

vaa   To save all data for the currently displayed areas, use the VAA (view save all) command. The syntax for this command is:

vaa *filename*

This saves all views of the data—including the individual count, inclusive, etc.— views with the percentage indications and histograms.

Both commands write profile data to *filename*. The filename can include path information. There is no default filename. If *filename* already exists, the command will overwrite the file with the new data.

Note that if the PROFILE window displays only a subset of the areas that are marked for profiling, data is saved *only for those areas that are displayed*. (For VAC, the currently displayed data will be saved for the displayed areas. For VAA, all data will be saved for the displayed areas.) If some areas are hidden and you want to save all the data, be sure to select View→Reset before saving the data to a file.

The file contents are in ASCII and are formatted in exactly the same manner as they are displayed (or would be displayed) in the PROFILE window. The general profiling-session information that is displayed in the COMMAND window is also written to the file.

Summary of Commands and Special Keys

This chapter summarizes the debugger's commands and special key sequences. It begins with a description of the various categories of debugger commands and then lists the commands that fall under these categories.

The main portion of this chapter is the alphabetical command reference. Each debugger command is listed with its syntax, applicable modes, its correspondence to a pulldown menu (if any), and a short description. The chapter ends with a summary of special keys and their functions in the debugging environment.

13.1 Functional Summary of Debugger Commands

This section summarizes the debugger commands according to these categories:

❏ **Changing modes.** These commands enable you to switch freely among the three debugging modes (auto, mixed, and assembly). You can also select these commands from the Mode pulldown menu.

❏ **Managing windows.** These commands enable you to select the active window and move or resize the active window. You can also perform these functions with the mouse.

❏ **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 also available on the Watch pulldown menu.

❏ **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.

❏ **Memory mapping.** These commands enable you to define the areas of target memory that the debugger can access. These commands are also available on the Memory pulldown menu.

❏ **Customizing the screen.** These commands allow you to customize the debugger display, then save and later reuse the customized displays. These commands are also available from the Color pulldown menu.

❏ **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.

❏ **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.

❏ **Profiling commands.** These commands enable you to collect execution statistics for your code. Commands can be entered from the pulldown menus or on the command line.

Changing modes

To do this	Use this command	See page
Put the debugger in assembly mode	asm	13-13
Put the debugger in auto mode for debugging C code	c	13-15
Put the debugger in mixed mode	mix	13-29

Managing windows

To do this	Use this command	See page
Make the active window as large as possible	zoom	13-52
Open the INSPECT window (XDS/22 only)	insp	13-25
Reposition the active window	move	13-30
Resize the active window	size	13-42
Select the active window	win	13-50

Performing system tasks

To do this	Use this command	See page
Associate a beeping sound with the display of error messages	sound	13-43
Change the current working directory from within the debugger environment	cd/chdir	13-16
Check the current version of the debugger	version	13-48
Clear all displayed information from the COMMAND window display area	cls	13-16
Conditionally execute debugger commands in a batch file	if/else/endif	13-25
Define your own command string	alias	13-12
Delete an alias definition	unalias	13-47
Delete *all* defined aliases	unalias *	13-47
Display a string to the COMMAND window while executing a batch file	echo	13-22
Enter any operating-system command or exit to a system shell	system	13-45
Execute commands from a batch file	take	13-46
Exit the debugger	quit	13-34

Performing system tasks (continued)

To do this	Use this command	See page
List the contents of the current directory or any other directory	dir	13-19
Load a saved BTT setup (XDS/22 only)	btt	13-15
Loop debugger commands in a batch file	loop/endloop	13-26
Name additional directories that can be searched when you load source files	use	13-47
Record the information shown in the COMMAND window display area	dlog	13-21
Reset the target system	reset	13-35
Store and save the information shown in the trace buffer to a file	tsave	13-46

Displaying and changing data

To do this	Use this command	See page
Change the default format for displaying data values	setf	13-41
Continuously display the value of a variable, register, or memory location within the WATCH window	wa	13-49
Delete a data item from the WATCH window	wd	13-49
Delete all data items from the WATCH window and close the WATCH window	wr	13-51
Display a different range of memory in the MEMORY window	mem	13-29
Display a pop-up MEMORY window	mem1,mem2, mem3	13-29
Display the values in an array or structure or display the value that a pointer is pointing to	disp	13-20
Evaluate a C expression without displaying the results	eval	13-23
Evaluate and display the result of a C expression	?	13-11
Show the type of a data item	whatis	13-50

Displaying files and loading programs

To do this	Use this command	See page
Display a specific C function	func	13-24
Display a text file in the FILE window	file	13-23
Display assembly language code at a specific address	dasm	13-19
Display C and/or assembly language code at a specific point	addr	13-12
Load an object file	load	13-26
Load only the object-code portion of an object file	reload	13-35
Load only the symbol-table portion of an object file	sload	13-43
Modify disassembly with the patch assembler	patch	13-32
Reopen the CALLS window	calls	13-16

Memory mapping

To do this	Use this command	See page
Add an address range to the memory map	ma	13-27
Delete an address range from the memory map	md	13-28
Enable or disable memory mapping	map	13-28
Initialize a block of memory	fill	13-23
Reset (delete all ranges) the memory map	mr	13-31
Save a block of memory to a system file	ms	13-31
Display a list of the current memory map settings	ml	13-30

Customizing the screen

To do this	Use this command	See page
Change the border style of any window	border	13-14
Change the command-line prompt	prompt	13-34
Change the screen colors and update the screen immediately	scolor	13-39
Change the screen colors, but don't update the screen immediately	color	13-17
Load and use a previously saved custom screen configuration	sconfig	13-40
Save a custom screen configuration	ssave	13-44

Running programs

To do this	Use this command	See page
Execute code in a function and return to the function's caller	return	13-36
Execute commands from a batch file	take	13-46
Halt the CPU after executing a RUNF command	halt	13-24
Reset the PC to the program entry point	restart	13-35
Reset the target system	reset	13-35
Reset the target system and run a program	rrun	13-36
Reset the target system and run the BTT and CPU simultaneously (while controlling the BTT independently)	rrunf	13-36
Run a program	run	13-37
Run the BTT and CPU simultaneously, while controlling the BTT independently	runf	13-38
Run a program up to a certain point	go	13-24
Run a program with benchmarking (count the number of CPU clock cycles consumed by the executing portion of code)	runb	13-37
Single-step through assembly language or C code	step	13-44
Single-step through assembly language or C code, one C statement at a time	cstep	13-18
Single-step through assembly language or C code one C statement at a time; step over function calls	cnext	13-17
Single-step through assembly language or C code; step over function calls	next	13-32
Wait for a hardware reset before running a program	wrun	13-51
Wait for a hardware reset before running the BTT and CPU	wrunf	13-52

Managing breakpoints

To do this	Use this command	See page
Add a breakpoint	ba	13-13
Delete a breakpoint	bd	13-13
Display a list of all the breakpoints that are set	bl	13-14
Reset (delete) all breakpoints	br	13-15

Profiling commands

All of the profiling commands can be entered from the pulldown menus. In many cases, using the pulldown menus is the easiest way to use some of these commands. For this reason and also because there are over 100 profiling commands, most of these commands are not described individually in this chapter (as the basic debugger commands are).

Listed below are some of the profiling commands that you might choose to enter from the command line instead of from a menu; these commands are also described in the alphabetical command summary. The remaining profiling commands are summarized in Section 13.4 on page 13-53.

To do this	Use this command	See page
Add a stopping point	sa	13-38
Delete a stopping point	sd	13-40
Delete all the stopping points	sr	13-43
List all the stopping points	sl	13-42
Reset the display in the PROFILE window to show all areas and the default set of data	vr	13-48
Resume a profiling session	pr	13-34
Run a full profiling session	pf	13-33
Run a quick profiling session	pq	13-33
Save all the profile data to a file	vaa	13-47
Save currently displayed profile data to a file	vac	13-48

13.2 How Menu Selections Correspond to Commands

The following sample screens illustrate the relationship of the basic debugger commands to the menu bar and pulldown menus.

Note:

Because the profiling environment supports over 100 profile-specific commands, it's not practical to show the commands associated with the profile menu choices.

Program execution commands

```
Run=F5      ──  RUN command
                (without a parameter)

Step=F8     ──  STEP command
                (without a parameter)

Next=F10    ──  NEXT command
                (without a parameter)
```

File/Load commands

```
Load
Load      ──────────  LOAD command
Reload    ──────────  RELOAD command
Symbols   ──────────  SLOAD command
REstart   ──────────  RESTART command
ReseT     ──────────  RESET command
File      ──────────  FILE command
Config    ──────────  CLOCK command
```

Breakpoint commands

```
Break
Add       ──────────  BA command
Delete    ──────────  BD command
Reset     ──────────  BR command
List      ──────────  BL command
```

Watch commands

```
         Watch                                  WA command
         Add                                    
         Delete                                 WD command
         Reset                                  
                                                WR command
```

Memory commands

```
         Memory                                 MA command
         Add                                    
         Delete                                 MD command
         Reset                                  MR command
         List                                   ML command
         Enable                                 
                                                MAP command
         Fill                                   FILL command
         Save                                   
                                                MS command
```
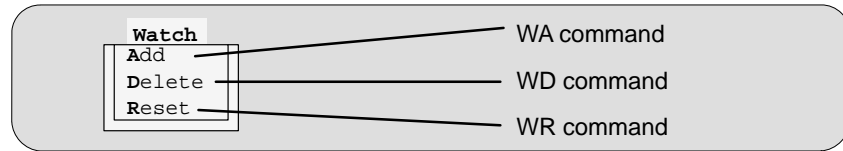
Screen-configuration commands

```
         Color                                  SCONFIG command
         Load                                   
         Save                                   SSAVE command
         Config                                 SCOLOR command
         Border                                 
         Prompt                                 BORDER command
                                                PROMPT command
```

Mode commands
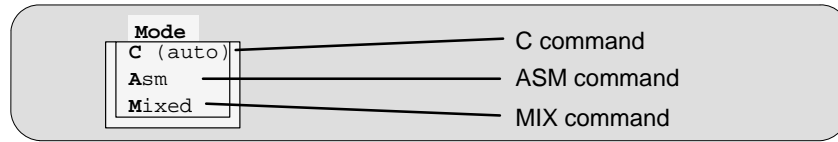
BTT menu and commands

Most of the selections on the BTT pulldown menu do not correspond to specific debugger commands. The illustration below shows which selections are associated with commands and lists the functions provided by the remaining entries.
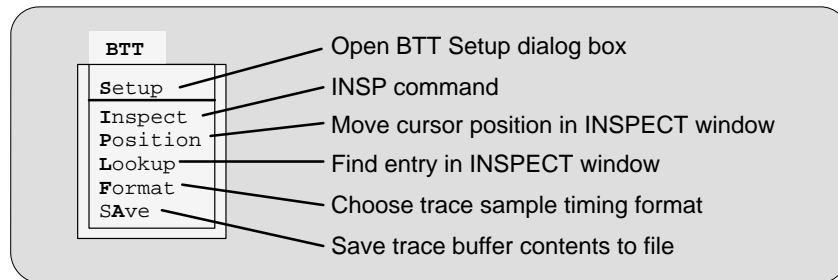
13.3 Alphabetical Summary of Debugger Commands

There are two debugger environments: the basic debugger environment and the profiling environment. Some debugger commands can be used in both environments; some can be used in only one of the environments. Each command description identifies the applicable environments for the command.

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

Environments   ☑ basic debugger   ☐ profiling

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 Specified Address*

Syntax   **addr**   *address*
                    addr   *function name*

Menu selection   none

Environments   ☑   basic debugger   ☐   profiling

Description   Use the ADDR command to display C code or the disassembly at a specific point.

❑ 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.

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

Environments   ☑   basic debugger   ☑   profiling

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 Mode

Syntax   **asm**

Menu selection   Mo**D**e→**A**sm

Environments   ☑ basic debugger   ☐ profiling

Description   The ASM command changes the current debugging mode to assembly mode. If you're already in assembly mode, the ASM command has no effect.

ba	Add Software Breakpoint

Syntax   **ba** *address*

Menu selection   **B**reak→**A**dd

Environments   ☑ basic debugger   ☐ profiling

Description   The BA command sets a software 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	Delete Software Breakpoint

Syntax   **bd** *address*

Menu selection   **B**reak→ **D**elete

Environments   ☑ basic debugger   ☐ profiling

Description   The BD command clears a software 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** | *List Software Breakpoints* |

Syntax　　　　　　　**bl**

Menu selection　　　　**B**reak→**L**ist

Environments　　　　　☑　basic debugger　　　　　　☐　profiling

Description　　　　　　The BL command provides an easy way to get a complete listing of all the software 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

Environments　　　　　☑　basic debugger　　　　　　☐　profiling

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	*Reset Software Breakpoint*

Syntax   **br**

Menu selection **B**reak→**R**eset

Environments   ☑ basic debugger   ☐ profiling

Description   The BR command clears all software breakpoints that are set.

btt	*Load Saved BTT Setup*	**XDS/22 only**

Syntax   **btt** *filename*

Menu selection none

Environments   ☑ basic debugger   ☐ profiling

Description   The BTT command loads a BTT setup that was saved by clicking on the <Save> field in the BTT Setup dialog box.

This restores the saved states and the actions that were defined for them.

c	*Enter Auto Mode*

Syntax   **c**

Menu selection Mo**D**e→**C** (auto)

Environments   ☑ basic debugger   ☐ profiling

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

Environments   ☑ basic debugger   ☐ profiling

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

Environments   ☑ basic debugger   ☑ profiling

Description   The CD or CHDIR command changes the current working directory from within the debugger. You can use relative pathnames as part of the *directory name*. If you don't use a *pathname,* the CD command displays the name of the current directory. When it is implemented with the USE command, CD 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 c:
cd d:\csource
cd c:\370tools
```

cls	*Clear Screen*

Syntax   **cls**

Menu selection   none

Environments   ☑ basic debugger   ☑ profiling

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)

Environments　　　☑　basic debugger　　　　　　☐　profiling

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 of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (the *Running code conditionally* discussion, page 7-17, discusses this in detail).

color　　　　　　　　*Change Screen Colors*

Syntax　　　　　　　**color**　*area name, attribute$_1$* [*,attribute$_2$* [*,attribute$_3$* [*,attribute$_4$*]]]

Menu selection　　none

Environments　　　☑　basic debugger　　　　　　☐　profiling

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)

Environments   ☑ basic debugger   ☐ profiling

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 of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (the *Running code conditionally* discussion, page 7-17, discusses this in detail).

dasm	*Display Disassembly at Specified Address*

Syntax   **dasm** *address*
                    dasm *function name*

Menu selection   none

Environments   ☑ basic debugger   ☑ profiling

Description   The DASM command displays code beginning at a specific point within the DISASSEMBLY window.

dir	*List Directory Contents*

Syntax   **dir** [*directory name*]

Menu selection   none

Environments   ☑ basic debugger   ☑ profiling

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

| **disp** | *Open DISP Window* |

Syntax   **disp**   *expression* [, *display format*]

Menu selection   none

Environments   ☑ basic debugger   ☐ profiling

Description   The DISP command opens a DISP window to display the contents of an array, structure, or pointer expression 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	**o**	Octal
c	ASCII character (bytes)	**p**	Valid address
d	Decimal	**s**	ASCII string
e	Exponential floating point	**u**	Unsigned decimal
f	Decimal floating point	**x**	Hexadecimal

The *display format* parameter can be used only when you are displaying a scalar type, an array of scalar type, or an 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.

dlog	*Record COMMAND Window Display*

Syntax   **dlog** *filename* [,{**a** | **w**}]
or
dlog close

Menu selection   none

Environments   ☑ basic debugger   ☑ profiling

Description   The DLOG command allows you to record the information displayed in the command window into a log file.

❏ To begin recording the information shown in the COMMAND window display area, use:

dlog *filename*

Log files can be executed by using the TAKE command. When you use DLOG to record the information from the COMMAND window display area into a log file called *filename*, the debugger automatically precedes all error or progress messages and command results with a semicolon to turn them into comments. This way, you can easily re-execute the commands in your log file by using the TAKE command.

❏ To end the recording session, enter:

dlog close ⏎

If necessary, you can write over existing log files or append additional information to existing files. The optional parameters of the DLOG command control how existing log files are used:

❏ **Appending to an existing file.** Use the **a** parameter to open an existing file to which to append the information in the display area.

❏ **Writing over an existing file.** Use the **w** parameter to open an existing file to write over the current contents of the file. Note that this is the default action if you specify an existing filename without using either the **a** or **w** options; you will lose the contents of an existing file if you don't use the append (a) option.

echo | *Echo String to Command Window*

Syntax   **echo** *string*

Menu selection   none

Environments   ☑   basic debugger   ☑   profiling

Description   The ECHO command displays *string* in the COMMAND window display area. This command works only in a batch file, and you can't use quote marks around the *string*. Note that any leading blanks in your command string are removed when the ECHO command is executed.

else | *Execute Alternative Debugger Commands*

Description   ELSE provides an alternative list of debugger commands in the IF/ELSE/ENDIF command sequence. See page 13-25 for more information about the IF/ELSE/ENDIF commands.

endif | *Terminate Conditional Sequence*

Description   ENDIF identifies the end of the IF/ELSE/ENDIF command sequence. See page 13-25 for more information about the IF/ELSE/ENDIF commands.

endloop | *Terminate Looping Sequence*

Description   ENDLOOP identifies the end of the LOOP/ENDLOOP command sequence. See page 13-26 for more information about the LOOP/ENDLOOP commands.

eval	Evaluate Expression

Syntax   **eval** *expression*
                      e *expression*

Menu selection   none

Environments   ☑ basic debugger   ☑ profiling

Description   The EVAL command evaluates an expression the same way 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

Environments   ☑ basic debugger   ☑ profiling

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

Environments   ☑ basic debugger   ☐ profiling

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 bytes that will be filled.

❏ The *data* is the value that the memory block will be filled with.

func	*Display Function*

Syntax   **func**   *function name*
                      func   *address*

Menu selection   none

Environments   ☑ basic debugger   ☑ profiling

Description   The FUNC command displays a specified C function in the FILE window. You can identify the function by its name or its address. Note that FUNC works the same way FILE works, but with 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

Environments   ☑ basic debugger   ☐ profiling

Description   The GO command executes code up to a specific point in your program. If you don't supply an *address* parameter, then GO acts like a RUN command without an *expression* parameter.

halt	*Halt Target System*	**XDS/22 Only**

Syntax   **halt**

Menu selection   none

Environments   ☑ basic debugger   ☐ profiling

Description   The HALT command halts both the BTT and CPU 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 be running the debugger in its normal mode of operation.

if/else/endif	*Conditionally Execute Debugger Commands*

Syntax
if *Boolean expression*
debugger command
debugger command
.
.
[**else**
debugger command
debugger command
.
.]
endif

Menu selection   none

Environments   ☑ basic debugger   ☑ profiling

Description   These commands allow you to conditionally execute debugger commands in a batch file. If the Boolean expression evaluates to true (1), the debugger executes the commands between the IF and the ELSE or ENDIF. Note that the ELSE portion of the command is optional.

The IF/ELSE/ENDIF conditional commands work with the following provisions:

❏ You can use IF/ELSE/ENDIF commands only in a batch file.

❏ You must enter each debugger command on a separate line in the batch file.

❏ You can't nest IF/ELSE/ENDIF commands within the same batch file.

insp	*Open INSPECT Window*	**XDS/22 only**

Syntax   **insp**

Menu selection   B**TT**→**I**nspect

Environments   ☑ basic debugger   ☐ profiling

Description   The INSP command opens the INSPECT window, which displays timing statistics and the contents of the trace buffer.

load	*Load Executable Object File*

Syntax   **load**   *object filename*

Menu selection   **L**oad→ **L**oad

Environments   ☑   basic debugger   ☑   profiling

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. If you don't supply an extension, the debugger looks for *filename*.out.

loop/endloop	*Loop Through Debugger Commands*

Syntax   **loop** *expression*
debugger command
debugger command
.
.
endloop

Menu selection   none

Environments   ☑   basic debugger   ☑   profiling

Description   The LOOP/ENDLOOP commands allow you to set up a looping situation in a batch file. These looping commands evaluate in the same method as in the run conditional command expression:

❑ If you use an *expression* that is not Boolean, the debugger evaluates the expression as a loop count.

❑ If you use a Boolean *expression*, the debugger executes the command repeatedly as long as the expression is true.

The LOOP/ENDLOOP commands work under the following conditions:

❑ You can use LOOP/ENDLOOP commands only in a batch file.

❑ You must enter each debugger command on a separate line in the batch file.

❏ You can't nest LOOP/ENDLOOP commands within the same batch file.

| **ma** | *Add Block to Memory Map* |

Syntax   **ma** *address, length, type*

Menu selection   **M**emory→**A**dd

Environments   ☑ basic debugger   ☑ profiling

Description   The MA command identifies valid ranges of target memory.

❏ The *address* parameter defines the starting address of a 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 of the range in bytes. 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 emulator memory	**R, ROM, EROM**
read-only external memory	**XROM**
read-only internal memory	**IROM**
read/write emulator memory	**RW, RAM, ERAM**
read/write external memory	**XRAM**
read/write internal memory	**IRAM**
read/write serial peripheral frame in emulator memory	**SEPER, SERW**
read/write serial peripheral frame in internal memory	**SIPER, SIRW**
read/write timer peripheral frame in emulator memory	**TEPER, TERW**
read/write timer peripheral frame in internal memory	**TIPER, TIRW**
no-access memory	**PROTECT**
EPROM control frame	**EPCTL**
program EPROM read-only emulator memory	**PEPROM**
data EPROM read-only emulator memory	**DEPROM**
custom EPROM read-only emulator memory	**CEPROM**
program EEPROM read-only emulator memory	**PEEPROM**
data EEPROM read-only emulator memory	**DEEPROM**
custom EEPROM read-only emulator memory	**CEEPROM**

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

Environments   ☑ basic debugger   ☑ profiling

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** | *Delete Block From Memory Map* |

Syntax   **md** *address*

Menu selection   **M**emory→**D**elete

Environments   ☑ basic debugger   ☑ profiling

Description   The MD command deletes a range of memory from the debugger's memory map. The *address* parameter identifies the starting address of the range of memory.

mem   *Modify MEMORY Window Display*

Syntax   **mem**[*#*]   *expression* [*, display format*]

Menu selection   none

Environments   ☑   basic debugger   ☐   profiling

Description   The MEM command identifies a new starting address for the block of memory displayed in a MEMORY window. The optional extension number (#) opens a pop-up MEMORY window allowing you to view a separate block of memory. 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 Mode*

Syntax   **mix**

Menu selection   Mo**D**e→**M**ixed

Environments   ☑   basic debugger   ☐   profiling

Description   The MIX command changes the current debugging mode to mixed mode. If you're already in mixed mode, the MIX command has no effect.

ml | *List Memory Map*

Syntax	**ml**
Menu selection	**M**emory→**L**ist
Environments	☑ basic debugger　　　☑ profiling
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.

move | *Move Active Window*

Syntax	**move** [*X position*, *Y position* [, *width*, *length*]]
Menu selection	none
Environments	☑ basic debugger　　　☑ profiling

Description　The MOVE command moves 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.

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. X positions are valid if the X position plus the window width in characters is less than or equal to the screen width in characters. Y positions are valid if the Y position plus the widow height is less than or equal to the screen height in lines.

For example, if the window is 10 characters wide and 5 lines high and the screen size is 80 x 25, the command **move 70, 20** would put the lower right-hand corner of the window in the lower right-hand corner of the screen. No X value greater than 70 or Y value greater than 20 would be valid in this example.

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 [↙] .

mr	*Reset Memory Map*

Syntax   **mr**

Menu selection   **M**emory→**R**eset

Environments   ☑   basic debugger   ☑   profiling

Description   The MR command resets the debugger's memory map by deleting all defined memory ranges from the map.

ms	*Save Memory Block to a File*

Syntax   **ms**   *address,length,filename*

Menu selection   **M**emory→**S**ave

Environments   ☑   basic debugger   ☑   profiling

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 of the block in bytes. This parameter can be any C expression.
❑ The *filename* is a system file. If you don't supply an extension, the debugger adds an .obj extension.

next   *Single-Step, Next Statement*

Syntax   **next**   [*expression*]

Menu selection   Next=**F10** (in disassembly or mixed mode)

Environments   ☑   basic debugger   ☐   profiling

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 of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (the *Running code conditionally* discussion, page 7-17, discusses this in detail).

patch   *Patch Assemble*

Syntax   **patch**   *address, assembly language instruction*

Menu selection   none

Environments   ☑   basic debugger   ☐   profiling

Description   The PATCH command allows you to patch-assemble disassembly statements. The *address* parameter identifies the address of the statement you want to change. The *assembly language instruction* parameter is the new statement you want to use at *address*. If you enter the command without parameters or with only the *address* parameter, the debugger will open a dialog box so that you can enter the remaining parameter(s).

pf	Profile, Full

Syntax   **pf** *starting point* [, *update rate*]

Menu selection   **P**rofile→**F**ull

Environments   ☐   basic debugger   ☑   profiling

Description   The PF command initiates a RUN and collects a full set of statistics on the defined areas between the *starting point* and the first-encountered stopping point. The *starting point* parameter can be a label, a function name, or a memory address.

The optional *update rate* parameter determines how often the PROFILE window will be updated. The *update rate* parameter can have one of these values:

Value	Description
0	This is the default. Statistics are not updated until the session is halted (although you can force an update by clicking the mouse in the window header). A "spinning wheel" character is shown to indicate that a profiling session is in progress.
≥**1**	Statistics are updated during the session. A value of **1** means that data is updated as often as possible.
<0	Statistics are not updated and the "spinning wheel" character is not displayed.

pq	Profile, Quick

Syntax   **pq** *starting point* [, *update rate*]

Menu selection   **P**rofile→**Q**uick

Environments   ☐   basic debugger   ☑   profiling

Description   The PQ command initiates a RUN command and collects a subset of the available statistics on the defined areas between the *starting point* and the first-encountered stopping point. PQ is similar to PF, except that PQ doesn't collect exclusive or exclusive max data.

The *update rate* parameter is the same as for the PF command.

pr	*Resume Profiling Session*

Syntax　　　　　　　　**pr**　[*clear data*　[, *update rate*]]

Menu selection　　　　**P**rofile→**R**esume

Environments　　　　☐　basic debugger　　　　☑　profiling

Description　　　　　The PR command resumes the last profiling session (initiated by PF or PQ), starting from the current program counter.

The optional *clear data* parameter tells the debugger whether or not it should clear out the previously collected data. The *clear data* parameter can have one of these values:

Value	Description
0	This is the default. The profiler will continue to collect data, adding it to the existing data for the profiled areas and to use the previous internal profile stacks.
nonzero	All previously collected profile data and internal profile stacks are cleared.

The *update rate* parameter is the same as for the PF and PQ commands.

prompt	*Change Command-Line Prompt*

Syntax　　　　　　　　**prompt**　*new prompt*

Menu selection　　　　**C**olor→**P**rompt

Environments　　　　☑　basic debugger　　　　☑　profiling

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

Environments　　　　☑　basic debugger　　　　☑　profiling

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

Environments   ☑   basic debugger   ☑   profiling

Description   The RELOAD command loads only an object file *without* loading its asso-
ciated symbol table. This is useful for reloading a program when target
memory has been corrupted. If you enter the RELOAD command without
specifying a filename, the debugger reloads the file that you loaded last.

reset	*Reset Target System*

Syntax   **reset**

Menu selection   **L**oad→Rese**T**

Environments   ☑   basic debugger   ☑   profiling

Description   The RESET command resets the target system.

restart	*Reset PC to Program Entry Point*

Syntax   **restart**
                      rest

Menu selection   **L**oad→R**E**start

Environments   ☑   basic debugger   ☑   profiling

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 pro-
gram into memory.)

return	Return to Function's Caller

Syntax   **return**
                     ret

Menu selection   none

Environments   ☑ basic debugger   ☐ profiling

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.

rrun	Reset and Run Code

Syntax   **rrun** [expression]

Menu selection   none

Environments   ☑ basic debugger   ☑ profiling

Description   The RRUN command resets the target system and then begins program execution. 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 7-17).

rrunf	Reset and Run Free

Syntax   **rrunf**

Menu selection   none

Environments   ☑ basic debugger   ☐ profiling

Description   The RRUNF command resets the target system and begins execution of the BTT independently from the CPU.

You can use the RRUNF command to begin execution of the BTT and CPU initially. However, after you have halted the BTT by pressing ESC, you must restart the BTT by using the RUNF command.

The HALT command stops a RRUNF; note that the debugger automatically executes a HALT when the debugger is invoked.

run	*Run Code*

Syntax   **run**   [*expression*]

Menu selection   Run=**F5**

Environments   ☑ basic debugger   ☐ profiling

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 7-17).

❑ 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	*Benchmark Code*	**XDS/22 only**

Syntax   **runb**

Menu selection   none

Environments   ☑ basic debugger   ☐ profiling

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 software 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 7.7, *Benchmarking,* on page 7-20.

runf	*Run Free*	**XDS/22 Only**

Syntax   **runf**

Menu selection   none

Environments   ☑   basic debugger   ☐   profiling

Description   The RUNF command simultaneously executes the BTT and CPU, while allowing the BTT to run independently from the CPU. With the CPU running, you can stop the BTT and perform operations such as dumping the contents of the trace buffer and reconfiguring the BTT. To stop the BTT, press ESC. Otherwise, *you must wait for the BTT or CPU to stop on their own.* While both the BTT and CPU are running, you do not have access to the command line.

After you have finished operations on the BTT, you can restart it by entering the RUNF command again.

You can use the RRUNF and WRUNF commands to begin execution of the BTT and CPU initially. However, after you have halted the BTT by pressing ESC, you must restart the BTT with the RUNF command.

The HALT command stops a RUNF; note that the debugger automatically executes a HALT when the debugger is invoked.

sa	*Add Stoppoint*	

Syntax   **sa** *address*

Menu selection   **S**top-points→**A**dd

Environments   ☐   basic debugger   ☑   profiling

Description   The SA command adds a stopping point at *address*. The *address* can be a label, a function name, or a memory address.

scolor   *Change Screen Colors*

Syntax   **scolor**   *area name*, *attribute₁* [, *attribute₂* [, *attribute₃* [, *attribute₄*]]]

Menu selection   **C**olor→**C**onfig

Environments   ☑ basic debugger   ☐ profiling

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			

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 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

Environments   ☑   basic debugger   ☑   profiling

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.

sd   *Delete Stoppoint*

Syntax   **sd**   *address*

Menu selection   **S**top-points→**D**elete

Environments   ☐   basic debugger   ☑   profiling

Description   The SD command deletes the stopping point at *address*.

setf *Set Default Data-Display Format*

Syntax **setf** [*data type*, *display format*]

Menu selection none

Environments ☑ basic debugger   ☐ profiling

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	float	ptr

❏ The *display format* parameter can be any of the following characters:

Parameter	Result	Parameter	Result
*	Default for the data type	o	Octal
c	ASCII character (bytes)	p	Valid address
d	Decimal	s	ASCII string
e	Exponential floating point	u	Unsigned decimal
f	Decimal floating point	x	Hexadecimal

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

Environments   ☑ basic debugger   ☑ profiling

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 in-teractively resize the window.

Valid values for the width and length depend on the screen size and the window position on the screen. If the window is in the upper left corner of the screen, the maximum size of the window is the same as the screen size minus one line. (The extra line is needed for the menu bar.) For example, if the screen size is 80 characters by 25 lines, the largest window size is 80 characters by 24 lines.

If a window is in the middle of the display, 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 4-26.

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 `↵` .

sl	*List Stoppoints*

Syntax   **sl**

Menu selection   **S**top-points→**L**ist

Environments   ☐ basic debugger   ☑ profiling

Description   The SL command lists all of the currently set stopping points.

sload   *Load Symbol Table*

Syntax   **sload**   *object filename*

Menu selection   **L**oad→**S**ymbols

Environments   ☑ basic debugger   ☑ profiling

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 Beeping*

Syntax   **sound   on** | **off**

Menu selection   none

Environments   ☑ basic debugger   ☐ profiling

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 (because you wouldn't see the error message). By default, sound is off.

sr   *Reset Stoppoints*

Syntax   **sr**

Menu selection   **S**top-points→**R**eset

Environments   ☐ basic debugger   ☑ profiling

Description   The SR command resets (deletes) *all* currently set stopping points.

ssave   *Save Screen Configuration*

Syntax   **ssave**   [*filename*]

Menu selection   **C**olor→**S**ave

Environments   ☑ basic debugger   ☐ profiling

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 or mixed mode)

Environments   ☑ basic debugger   ☐ profiling

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 (the *Running code conditionally* discussion, page 7-17, discusses this in detail).

system	*Enter DOS Command*

Syntax   **system**   [*DOS command* [, *flag*]]

Menu selection   none

Environments   ☑   basic debugger   ☑   profiling

Description   The SYSTEM command allows you to enter DOS 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 DOS 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:

exit   ⏎

If you prefer, you can supply the DOS command as a parameter to the SYS-TEM 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

Environments   ☑ basic debugger   ☑ profiling

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.

❑ 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.

tsave	*Store and Save the Trace Buffer*	**XDS/22 Only**

Syntax   **tsave** *filename*

Menu selection   none

Environments   ☑ basic debugger   ☐ profiler

Description   The TSAVE command stores and saves the information shown in the trace buffer to a file called *filename*.

unalias	*Delete Alias Definition*

Syntax   **unalias**   *alias name*
                      unalias   *

Menu selection   none

Environments   ☑ basic debugger   ☑ profiling

Description   The UNALIAS command deletes defined aliases.

❏ To delete a **single alias**, enter the UNALIAS command with an alias name. For example, to delete an alias named NEWMAP, enter:

unalias NEWMAP ⏎

❏ To delete **all aliases**, enter an asterisk instead of an alias name:

unalias * ⏎

Note that the * symbol *does not* work as a wildcard.

use	*Use New Directory*

Syntax   **use**   *directory name*

Menu selection   none

Environments   ☑ basic debugger   ☑ profiling

Description   The USE command allows you to name an additional directory that the debugger can search when looking for source files. You can specify only one directory at a time.

If you enter the USE command without specifying a directory name, the debugger lists all of the current directories.

vaa	*Save All Profile Data to a File*

Syntax   **vaa**   *filename*

Menu selection   **V**iew→**S**ave→**A**ll views

Environments   ☐ basic debugger   ☑ profiling

Description   The VAA command saves all statistics collected during the current profiling session. The data is stored in a system file.

vac	*Save Displayed Profile Data to a File*

Syntax   **vac** *filename*

Menu selection   **V**iew→**Sa**ve→**C**urrent view

Environments   ☐ basic debugger   ☑ profiling

Description   The VAC command saves all statistics currently displayed in the PROFILE window. (Statistics that aren't displayed aren't saved.) The data is stored in a system file.

version	*Display the Current Debugger Version*

Syntax   **version**

Menu selection   none

Environments   ☑ basic debugger   ☑ profiler

Description   The VERSION command displays the debugger's copyright date and the current version number of the debugger, silicon, XDS, and BTT.

vr	*Reset PROFILE Window Display*

Syntax   **vr**

Menu selection   **V**iew→**R**eset

Environments   ☐ basic debugger   ☑ profiling

Description   The VR command resets the display in the PROFILE window so that all marked areas are listed and the statistics are displayed with the default labels and in the default sort order.

wa	Add Item to WATCH Window

Syntax   **wa** *expression* [,[*label*], *display format*]
                          wa **expression* [,[*label*], *display format*]

Menu selection   **W**atch→**A**dd

Environments   ☑ basic debugger   ☐ profiling

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. In an assembly language program, you can use a symbol name as the expression parameter. To watch the contents of a symbol name, you must precede the expression parameter an asterisk (refer to *Displaying Data in a WATCH Window*, on page 8-14). 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

wd	Delete Item From WATCH Window

Syntax   **wd** *index number*

Menu selection   **W**atch→**D**elete

Environments   ☑ basic debugger   ☐ profiling

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	*Find Data Type*

Syntax   **whatis** *symbol*

Menu selection   none

Environments   ☑ basic debugger   ☐ profiling

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

Environments   ☑ basic debugger   ☑ profiling

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   *Reset WATCH Window*

Syntax   **wr**

Menu selection   **W**atch→**R**eset

Environments   ☑ basic debugger   ☐ profiling

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

wrun   *Wait and Run*

Syntax   **wrun** [*expression*]

Menu selection   none

Environments   ☑ basic debugger   ☑ profiling

Description   The WRUN command waits for the emulator to ascert a reset in the target
                          system and then it executes an entire program. The command's behavior de-
                          pends 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 7-17).

wrunf	*Wait and Run Free*

Syntax   **wrunf**

Menu selection   none

Environments   ☑ basic debugger   ☐ profiling

Description   The WRUNF command waits for the debugger to reset the target system, and then begins execution of the BTT independently from the CPU.

You can use the WRUNF command to begin execution of the BTT and CPU initially. However, after you have halted the BTT by pressing ESC, you must restart the BTT by using the RUNF command.

The HALT command stops a WRUNF; note that the debugger automatically executes a HALT when the debugger is invoked.

zoom	*Zoom Active Window*

Syntax   **zoom**

Menu selection   none

Environments   ☑ basic debugger   ☑ profiling

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.

13.4 Summary of Profiling Commands

The following tables summarize the profiling commands that are used for marking, enabling, disabling, and unmarking areas and for changing the display in the PROFILE window. These commands are easiest to use from the pulldown menus, so they are not included in the alphabetical command summary. The syntaxes for these commands are provided here so that you can include these commands in batch files.

Table 13–1. Marking Areas

To mark this area	C only		Disassembly only
Lines			
By line number, address	**MCLE**	*filename, line number*	**MALE** *address*
All lines in a function	**MCLF**	*function*	**MALF** *function*
Ranges			
By line numbers	**MCRE**	*filename, line number, line number*	**MARE** *address, address*
Functions			
By function name	**MCFE**	*function*	not applicable
All functions in a module	**MCFM**	*filename*	
All functions everywhere	**MCFG**		

Table 13–2.Disabling Marked Areas

To disable this area	C only		Disassembly only		C *and* disassembly	
Lines						
❏ By line number, address	**DCLE**	*filename, line number*	**DALE**	*address*	not applicable	
❏ All lines in a function	**DCLF**	*function*	**DALF**	*function*	**DBLF**	*function*
❏ All lines in a module	**DCLM**	*filename*	**DALM**	*filename*	**DBLM**	*filename*
❏ All lines everywhere	**DCLG**		**DALG**		**DBLG**	
Ranges						
❏ By line numbers, addresses	**DCRE**	*filename, line number*	**DARE**	*address*	not applicable	
❏ All ranges in a function	**DCRF**	*function*	**DARF**	*function*	**DBRF**	*function*
❏ All ranges in a module	**DCRM**	*filename*	**DARM**	*filename*	**DBRM**	*filename*
❏ All ranges everywhere	**DCRG**		**DARG**		**DBRG**	
Functions						
❏ By function name	**DCFE**	*function*	not applicable		not applicable	
❏ All functions in a module	**DCFM**	*filename*			**DBFM**	*filename*
❏ All functions everywhere	**DCFG**				**DBFG**	
All areas						
❏ All areas in a function	**DCAF**	*function*	**DAAF**	*function*	**DBAF**	*function*
❏ All areas in a module	**DCAM**	*filename*	**DAAM**	*filename*	**DBAM**	*filename*
❏ All areas everywhere	**DCAG**		**DAAG**		**DBAG**	

Table 13–3. Enabling Disabled Areas

To enable this area	C only		Disassembly only		C *and* disassembly	
Lines						
By line number, address	**ECLE**	*filename***,** *line number*	**EALE**	*address*	not applicable	
All lines in a function	**ECLF**	*function*	**EALF**	*function*	**EBLF**	*function*
All lines in a module	**ECLM**	*filename*	**EALM**	*filename*	**EBLM**	*filename*
	ECLG		**EALG**		**EBLG**	
All lines everywhere						
Ranges						
By line numbers,addresses	**ECRE**	*filename***,** *line number*	**EARE**	*address*	not applicable	
	ECRF	*function*	**EARF**	*function*	**EBRF**	*function*
All ranges in a function	**ECRM**	*filename*	**EARM**	*filename*	**EBRM**	*filename*
	ECRG		**EARG**		**EBRG**	
All ranges in a module						
All ranges everywhere						
Functions						
By function name	**ECFE**	*function*	not applicable		not applicable	
All functions in a module	**ECFM**	*filename*			**EBFM**	*filename*
	ECFG				**EBFG**	
All functions everywhere						
All areas						
All areas in a function	**ECAF**	*function*	**EAAF**	*function*	**EBAF**	*function*
All areas in a module	**ECAM**	*filename*	**EAAM**	*filename*	**EBAM**	*filename*
	ECAG		**EAAG**		**EBAG**	
All areas everywhere						

Table 13–4. Unmarking Areas

To unmark this area	C only	Disassembly only	C *and* disassembly
Lines			
By line number, address	**UCLE** *filename, line number*	**UALE** *address*	not applicable
All lines in a function	**UCLF** *function*	**UALF** *function*	**UBLF** *function*
All lines in a module	**UCLM** *filename*	**UALM** *filename*	**UBLM** *filename*
All lines everywhere	**UCLG**	**UALG**	**UBLG**
Ranges			
By line numbers, addresses	**UCRE** *filename, line number*	**UARE** *address*	not applicable
All ranges in a function	**UCRF** *function*	**UARF** *function*	**UBRF** *function*
All ranges in a module	**UCRM** *filename*	**UARM** *filename*	**UBRM** *filename*
All ranges everywhere	**UCRG**	**UARG**	**UBRG**
Functions			
By function name	**UCFE** *function*	not applicable	not applicable
All functions in a module	**UCFM** *filename*		**UBFM** *filename*
All functions everywhere	**UCFG**		**UBFG**
All areas			
All areas in a function	**UCAF** *function*	**UAAF** *function*	**UBAF** *function*
All areas in a module	**UCAM** *filename*	**UAAM** *filename*	**UBAM** *filename*
All areas everywhere	**UCAG**	**UAAG**	**UBAG**

Table 13–5.Changing the PROFILE Window Display

(a) Viewing specific areas

To view this area	C only		Disassembly only	C *and* disassembly
Lines				
By line number, address	**VFCLE**	*filename***,** *line number*	**VFALE** *address*	not applicable
All lines in a function	**VFCLF**	*function*	**VFALF** *function*	**VFBLF** *function*
All lines in a module	**VFCLM**	*filename*	**VFALM** *filename*	**VFBLM** *filename*
	VFCLG		**VFALG**	**VFBLG**
All lines everywhere				
Ranges				
By line numbers,addresses	**VFCRE**	*filename***,** *line number*	**VFARE** *address*	not applicable
	VFCRF	*function*	**VFARF** *function*	**VFBRF** *function*
All ranges in a function	**VFCRM**	*filename*	**VFARM** *filename*	**VFBRM** *filename*
All ranges in a module	**VFCRG**		**VFARG**	**VFBRG**
All ranges everywhere				
Functions				
By function name	**VFCFE**	*function*	not applicable	not applicable
All functions in a module	**VFCFM**	*filename*		**VFBFM** *filename*
	VFCFG			**VFBFG**
All functions everywhere				
All areas				
All areas in a function	**VFCAF**	*function*	**VFAAF** *function*	**VFBAF** *function*
All areas in a module	**VFCAM**	*filename*	**VFAAM** *filename*	**VFBAM** *filename*
	VFCAG		**VFAAG**	**VFBAG**
All areas everywhere				

(b) Viewing different data

To view this information	Use this command
Count	**VDC**
Inclusive	**VDI**
Inclusive, maximum	**VDN**
Exclusive	**VDE**
Exclusive, maximum	**VDX**
Address	**VDA**
All	**VDL**

(c) Sorting the data

To sort on this data	Use this command
Count	**VSC**
Inclusive	**VSI**
Inclusive, maximum	**VSN**
Exclusive	**VSE**
Exclusive, maximum	**VSX**
Address	**VSA**
Data	**VSD**

13.5 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
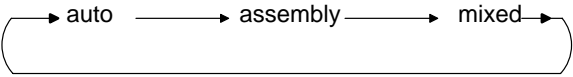
Editing text on the command line

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	CONTROL H or BACK SPACE
Move forward through text without erasing characters	CONTROL 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
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 these function keys
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 display area of the COMMAND window	

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 the BTT menu	ALT T
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, DISP, or additional MEMORY window (the window must be active before you can close it)	F4

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	CONTROL PAGE UP
DISP windows only: Scroll down through an array of structures	CONTROL 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 these 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	

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, thus reducing 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. If you're an experienced C programmer, skip Section 14.1.

Because the C expressions you'll use are parameters to debugger commands, some language features may be inappropriate. Therefore, Section 14.2 covers specific implementation issues (including necessary limitations and additional features) related to using C expressions as command parameters.

14.1 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 should at least be familiar with the rules governing C expressions. A helpful reference is **The C Programming Language** (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.

14.2 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.)

❏ 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, however, the function must be somewhere in the current call stack. 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 following 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.

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 can 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
*R5
*(R2 + 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, in the expression:

```
((struct STR *)10)->field
```

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.

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) Initializes the emulator and BTT, and resets the memory map.

5) Looks for the screen configuration file:

   init.clr file.

   (The debugger searches for the screen configuration file in directories named with D_DIR.)

6) Initializes the debugger screen and windows but initially displays only the COMMAND window.

7) Finds the batch file that defines your memory map by searching in directories named with D_DIR. The debugger expects this file to set up the memory map and follows these steps to look for the batch file:

   a) When you invoke the debugger, it checks to see if you've used the –t debugger option. If it finds the –t option, the debugger reads and executes the specified file.

   b) If you don't use the –t option, the debugger looks for

   c) a file called *init.cmd*. If the debugger finds the file, it reads and executes the file.

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.

Setting Up the Clock
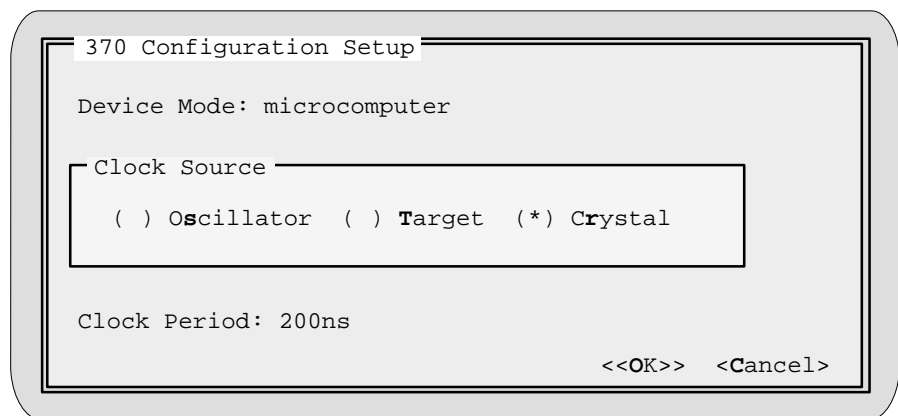
For proper emulation you need to be using the same operating frequency as your end application. Therefore, if you want to use a clock source other than the default crystal, you must setup the '370 debugger to match your new clock source. The debugger has three ways of generating the clock, including:

❏ An **Oscillator** clock generated through an oscillator circuit on the emulator.

❏ A **Target** clock generated from the target system. Note that your external clock *must* be CMOS compatible; a crystal will not work because the signal is not strong enough to reach the emulator board.

❏ A **Crystal** clock generated through a crystal inside the emulator.

Refer to your installation guide for instructions on installing the different clock sources.

To select and monitor a particular clock, you can use the CONFIG dialog box on the LOAD menu. The CONFIG dialog box looks like this:

```
370 Configuration Setup

Device Mode: microcomputer

Clock Source

  ( ) Oscillator  ( ) Target  (*) Crystal


Clock Period: 200ns

                              <<OK>>  <Cancel>
```

To select a clock through the CONFIG dialog box:

1) Point to the clock source you want to select.

2) Now click the left button. An asterisk is displayed next to the clock source you enabled.

Another way to select a particular clock is to use the alias command, *clock*. For example, to select the oscillator clock, you would enter:

```
clock=oscillator
```

Alternately, you can select the clock by assigning the appropriate value to the pseudoregister. The values for the oscillator, target, and crystal clocks are 0, 1, and 2, respectively. Therefore, to select the oscillator clock using the appropriate value, enter:

```
?clock = 0
```

Hint: To be sure the debugger set the clock source according to the new clock, pay attention to the *clock period*. The debugger computes the clock period from the clock source you selected as a way to verify the setup.

Note:

When programming in C, do not use a variable named CLOCK.

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.

C.1 Associating Sound With Error Messages

You can associate a beeping sound with the display of an error message. 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.

C.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 C.3 (page C-22).

')' 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 C.3 (page C-22).

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. 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 breakpoints are set in your program. Use the BR command to delete all breakpoints, or use the BD command to delete individual 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 may be loaded. Or you might want to relink your program and link in fewer modules at a time.

Cannot allocate system 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 may be loaded. Or you might want to relink your program and link in fewer modules at a time.

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:\370tools, not just 370tools).

Cannot detect target power

Description   This hardware error occurs after resetting the emulator (with *emurst*). Follow the steps described below and then restart your emulator.

Action

❑   Check the emulator board to be sure it is installed snugly.

❑   Check the cable connecting your emulator and target system to be sure it is not loose.

❑   Check the power to be sure it is on.

❑   Check your target board to be sure it is getting the correct voltage.

❑   Check your emulator scan path to be sure it is uninterrupted.

❑   Ensure that your serial port is set correctly. See the section on *Identifying the port address (–p option)*, page 1-14.

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 execute command

Description   The command you attempted to execute cannot be executed while the emulator or BTT is running.

Action   Halt the emulator or BTT *before* you attempt to execute this command.

Cannot find/open initialization file

Description   The debugger can't find the init.cmd file.

Action   Be sure that the init.cmd is in the appropriate directory. If it isn't, copy it from the debugger product diskette. If the file is already in the correct directory, verify that the D_DIR environment variable is set up to identify the directory. See *Setting Up the Debugger Environment* in the installation guide.

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.bat or config.sys file, then invoke the debugger again.

Cannot initialize target system

Description   This error occurs while you are invoking the debugger. Any combination of events may cause this error to occur.

Action   ❏   Check the cable connecting the emulator to the target system to be sure it is not loose.

        ❏   Check the cable connecting the emulator to the PC to be sure it is not loose.

        ❏   Check the power to be sure it is on.

        ❏   Check your SET D_OPTIONS in the autoexec.bat file or initdb.bat file, depending on which operating system you are using. The *port address* (or –p option) is not set correctly. If you didn't note the I/O switch settings, use a trial-and-error approach to find the correct setting.

        ❏   Ensure that your serial port is set correctly. See the section on *Identifying the port address (–p option)*, page 1-14.

Cannot map into reserved memory: ?

Description   The debugger tried to access to unconfigured/reserved/non-existent memory.

Action   Remap the reserved memory accesses. See the discussion on memory mapping in the *TMS370 Family XDS/22 User's Guide* for more information.

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 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, DISP, and additional MEMORY windows. To close the WATCH window, enter WD. To close the CALLS window, a DISP window, or a MEMORY window make the desired window active and press F4 .

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 cl370 again to create an executable object file).

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.bat or initdb.bat file, then invoke the debugger again.

Cannot reset 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 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.

Action   Check your memory map. If the address that you wanted to breakpoint wasn't in ROM, see Section C.4 (page C-22).

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 C.3 (page C-22).

Command "*cmd***" not found**

Description   The debugger didn't recognize the command that you typed.

Action   Re-enter the correct command. Refer to Chapter 13 or the Quick Reference Card for a list of valid debugger commands.

Command timed out, emulator busy

Description   There is a problem with the target system.

Action   See Section C.4 (page C-22).

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 overlap. 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.

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.

E

EEPROM control register not defined

Description   You attempted to define a block of memory for the EEPROMs/EPROMs without defining the EEPROM/EPROM control register.

Action   Define the EEPROM/EPROM control register *before* defining memory for the EEPROM/EPROM.

Emulator error

Description   A low-level hardware error has occurred.

Action   See Section C.4 (page C-22).

Emulator memory error

Description   There is a problem with your emulator memory.

Action   Be sure the emulator board not damaged and is placed snugly in your PC. Be sure you do not have a control address conflict.

Emulator translation error

Description   There is a problem with the target system.

Action   See Section C.4 (page C-22).

Error in expression

Description   This is an expression error.

Action   See Section C.3 (page C-22).

Exception encountered

Description   The debugger received an unexpected communication from the emulator.

Action   Reset the emulator.

Execution error

Description   A low-level hardware error occurred while you were trying to run the device.

Action   Be sure the device is not reset. See Section C.3 (page C-22).

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 C.3 (page C-22).

Function required

Description   The parameter for the FUNC command entered was not the name of a function in the program that is loaded.

Action   Re-enter the FUNC command with a valid function name.

I

Illegal addressing mode

Description   An illegal '370 addressing mode was encountered.

Action   Refer to Table 11–1, *Number of Actions Allowed Per State*, in section 11.4 (page 11-12) for valid addressing modes.

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 C.3 (page C-22).

Illegal left hand side of assignment

Description   This is an expression error—the lefthand side of an assignment expression doesn't meet C language assignment rules.

Action   See Section C.3 (page C-22).

Illegal memory access

Description   The debugger tried to access unconfigured/reserved/nonexistent memory.

Illegal opcode

Description   An invalid '370 instruction was encountered.

Action   Modify your source code or reset the emulator.

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 C.3 (page C-22).

Illegal pointer math

Description   This is an expression error—some types of pointer math are not valid in C expressions.

Action   See Section C.3 (page C-22).

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 C.3 (page C-22).

Illegal structure reference

Description   This is an expression error—either the item being referenced as a structure is not a structure, or you are attempting to reference a nonexistent portion of a structure.

Action   See Section C.3 (page C-22).

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 C.3 (page C-22).

Illegal use of void expression

Description   This is an expression error—the expression parameter does not meet the C language rules.

Action   See Section C.3 (page C-22).

Integer not allowed

Description   This is an expression error—the command did not accept an integer as a parameter.

Action   See Section C.3 (page C-22).

Invalid address

Description   Either the defined memory block for the peripheral frame fell outside of the range 0x1010–0x1100, or the starting and ending addresses of the memory block are not multiples of 0x10.

Action   ❑ Check to be sure the starting and ending addresses fall within the range 0x1010–0x1100. (This pertains only to the peripheral frame.)

❑ Check to be sure the starting and ending addresses are multiples of 0x10.

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 13.

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 10–2 (page 10-3).

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 10–1 (page 10-2).

Invalid length for EEPROM control register

Description   The length of the peripheral frame defined while mapping the EEPROM control register is invalid.

Action   Define a length of ten for the peripheral frame.

Invalid map type

Description   The EEPROM/EPROM or EEPROM/EPROM control frame you attempted to define already exists.

Action   Delete the existing EEPROM/EPROM or EEPROM/EPROM control frame and redefine the map type.

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, EROM	(read-only emulator memory)
XROM	(read-only external memory)
IROM	(read-only internal memory)
RW, RAM, ERAM	(read/write emulator memory)
XRAM	(read/write external memory)
IRAM	(read/write internal memory)
SERW, SEPER	(read/write serial peripheral frame in emulator memory)
SIRW, SIPER	(read/write serial peripheral frame in internal memory)
TERW, TEPER	(read/write timer peripheral frame in emulator memory)
TIRW, TIPER	(read/write timer peripheral frame in internal memory)
PROTECT	(no-access memory)
EPCTL	(EPROM control frame)
PEPROM	(program EPROM read-only emulator memory)
DEPROM	(data EPROM read-only emulator memory)
CEPROM	(custom EPROM read-only emulator memory)
PEEPROM	(program EEPROM read-only emulator memory)
DEEPROM	(data EEPROM read-only emulator memory)
CEEPROM	(custom EEPROM read-only emulator memory)

Invalid object file

Description   Either the file specified with the LOAD, SLOAD, or RELOAD command 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 dspcl 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 dspcl.

Invalid register

Description   This is an internal error.

Action   Shutdown the debugger and restart it. If the problem recurs, call the hotline.

Invalid scan path

Description   This is an internal error.

Action   Shutdown the debugger and restart it. If the problem recurs, call the hotline.

Invalid state

Description   Your target system is in an invalid state.

Action   Reset the emulator.

Invalid target revision

Description   The debugger attempted to debug an unknown revision of the '370 processor.

Action   Your debugger needs to be updated.

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 was typed in instead of a watch index.

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're finished, you *must* press ESC or ⏎.

❏ If you prefer to use the MOVE command with parameters, 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're finished, you *must* press ESC or ⏎.

❏ If you prefer to use the SIZE command with parameters, 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 C.4 (page C-22).

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 C.4 (page C-22).

Lost target power or 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 C.4 (page C-22).

Lval required

Description   This is an expression error—an assignment expression was entered that requires a legal left-hand side.

Action   See Section C.3 (page C-22).

M

Memory access error at *address*

Description   Either the processor is receiving a bus fault, or there are problems with target system memory.

Action   See Section C.4 (page C-22).

Memory access outside valid range: *address*

Description   Your program tried to access unmapped memory.

Action   Check your memory map.

Memory map table full

Description   Too many blocks have been added to the memory map. This rarely happens 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.

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.

No breakpoint at address

Description   This is an internal error.

Action   Shutdown the debugger and restart it. If the problem recurs, call the hotline.

P

Pointer not allowed

Description   This is an expression error.

Action   See Section C.3 (page C-22).

Processor access timeout at address

Description   There is a problem with your target system.

Action   See Section C.4 (page C-22).

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

Description   Either the processor is receiving a bus fault, or there are problems with target-system memory.

Action   See Section C.4 (page C-22).

RESET encountered

Description   The debugger received a hardware reset from another device on the system.

Action   Reset the emulator *if* this problem recurs frequently.

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 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 C.3 (page C-22).

Structure member name required

Description   This is an expression error—a symbol name followed by a period but no member name.

Action   See Section C.3 (page C-22).

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 C.3 (page C-22).

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 breakpoints are set in your program. Use the BR command to delete all breakpoints, or use the BD command to delete individual breakpoints.

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*.

U

Unable to read EMULATOR ID

Description The debugger was unable to read the emulator identification.

Action Reset the emulator.

Unable to send command packet

Description   The debugger was unable to send the command you entered to the emulator.

Action   Re-enter the command; if this does not work, reset the emulator.

User halt

Description   The debugger halted program execution because you pressed the ESC key.

Action   None required; this is normal debugger behavior.

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**SSEMBLY	**F**ILE	**W**ATCH
PROFILE	**I**NSPECT		

C.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.

C.4 Additional Instructions for Hardware Errors

If you continue to receive the messages that send you to this section, this indicates persistent hardware problems.

❏ If a bus fault occurs, the emulator may not be able to access memory.

❏ The '370 must be reset before you can use the emulator. Most target systems reset the '370 at power-up; your target system may not be doing this.

Glossary

A

action: A function performed by the BTT. Supported actions include hardware breakpoints, event counting, collection of trace samples, jumps to BTT states, and collection of timing statistics. Actions are not performed unless they meet predefined conditions.

active window: The window that is currently selected for moving, sizing, editing, closing, or some other function.

address-only state mode: A BTT mode that allows you to qualify actions based on address values but prevents you from qualifying actions based on data values.

address-and-data state mode: A BTT mode that allows you to qualify actions based on a combination of address and data values. This mode limits the number of actions that can be defined for a state.

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 showing assembly language code in the DISASSEMBLY window and C code in the FILE window, depending on what type of code is currently running.

B

batch file: One of two different types of files. One type 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.

BP/event: An action performed by the BTT when predefined conditions are met. Depending upon the conditions you have defined, the BTT will perform a hardware breakpoint, or it will decrement the event counter.

breakpoint: A point within your program where execution will halt because of a previous request from you.

BTT: *Breakpoint, trace, and timing*. A set of features supported by the BTT board (included with the XDS/22 emulation system). BTT features allow you to set hardware breakpoints, collect trace samples, and perform timing analysis.

BTT setup: A collection of information that you have defined for using the BTT. The setup includes conditions defined for actions, the actions that are defined for each state, and the global settings; a setup can be saved and reused.

C

C: A high-level, general-purpose programming language useful for writing compilers and operating systems and for programming microprocessors.

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.

CEEPROM: *Custom electrically erasable programmable read-only memory*.

CEPROM: *Custom erasable programmable read-only memory*.

children: Additional windows opened for aggregate types that are members of a parent aggregate type displayed in an existing DISP window.

cl370:   A shell utility that invokes the TMS370 compiler, assembler, and linker to create an execution object file version of your program.

click: To press and release a mouse button without moving the 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.

clock: An aliased command that is used to select the clock source (target, crystal, oscillator). Refer to Appendix B *Setting Up the Clock*.

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 TMS370 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: A block-shaped cursor that identifies the current character position on the command line.

COMMAND window: A window that provides an area for you to enter commands and for the debugger to echo command entry, show command output, and list progress or error messages.

CPU window: A window that displays the contents of '370 on-chip registers, including the register and peripheral files.

current-field cursor: A screen 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.

*Change the **dbrst** ⟶
entry to match the
utility in your book.*

dbrst: A utility that resets the SWDS.

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 '370 programs running on an emulator or application board.

DEEPROM: *Data electrically erasable programmable read-only memory.*

delay counter: A global BTT setting that defines how many trace samples can be collected between the time when a BP/event is detected and the time when the CPU and BTT are halted.

DEPROM: *Data erasable programmable read-only memory.*

disassembly: Assembly language code formed from the reverse-assembly of the contents of memory.

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.

EEPROM: *Electrically erasable programmable read-only memory.*

EPROM: *Erasable programmable read-only memory.*

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 '370 processor that is on the target system.

end state: A global BTT setting that defines the last state in a sequence of states. By default, the end state is 0.

environment variable: A special system symbol that the debugger uses for finding directories or obtaining debugger options.

event counter: A counter associated with a state that can count down the number of times BP/event conditions are qualified. The default event-counter value is 0; a hardware breakpoint cannot occur until the event counter reaches 0.

external-signal probe: A part of the BTT board that can be connected to external signals so that you can monitor the signals or use the values of the signals for qualifying actions.

Edit as necessary. $\Longrightarrow$ **EVM:** *Evaluation Module.* A development tool that lets you execute and debug applications programs by using the 'C5x debugger.

Change as neces- $\Longrightarrow$
sary. **evmrst:** A utility that resets the EVM.

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

global settings: Settings that control global operation of the BTT board. These settings include the delay counter, max trace, end state, loop counter, and time-out timer values.

H

hardware breakpoint: A hardware function performed by the BTT board; halts both the BTT board and the '370 CPU.

I

IAQ: Instruction acquisition cycle.

init.cmd: A batch file that contains debugger commands. If this file isn't present when you first invoke the debugger, then all memory is invalid.

initdb.bat: A batch file created to contain DOS commands to set up the debugger environment.

INSPECT window: A window that displays the contents of the trace buffer and timing statistics collected for timer 1 and timer 2.

ISA: *Industry Standard Architecture*. A subset of the EISA standard.

J

jump: An action performed when the BTT is used and when predefined conditions are met; the BTT jumps to the next sequential state.

L

loop counter: A global BTT setting that defines the number of times the BTT will sequence through states before taking a hardware breakpoint.

M

masking: To ignore specific bits within an address, data, or external-signal value.

max trace: A global BTT setting that defines the maximum number of trace samples that the BTT can collect. The default value of max trace is 0, which allows the BTT to collect all trace samples that qualify. A nonzero max-trace value acts as a maximum limit.

memory map: A map of memory space 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: A block-shaped cursor that tracks mouse movements over the entire display.

MR: Memory read cycle.

MW: Memory write cycle.

N

normal trace mode: A type of trace mode that allows only those cycles meeting predefined conditions to qualify as trace samples.

P

PC: Personal computer or program counter, 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.

PEEPROM: *Program electrically erasable programmable read-only memory*.

PEPROM: *Program erasable programmable read-only memory*.

point: To move the mouse cursor until it overlays the desired object on the screen.

point timer: (1) A timer that can be started and stopped when predefined addresses are accessed. (2) An action performed by the BTT when predefined conditions are met.

port address: The serial port that the debugger uses for communicating with the emulator or the applications board. The port address is selected, depending on which communication port the debugger is attached to.

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.

Q

qualifying: The process of matching bus activity to predefined conditions. Matching bus activity and conditions *qualifies* an action so that it can be performed by the BTT.

R

range timer: (1) A timer that can be started and stopped on any set of conditions. (2) An action performed by the BTT when predefined conditions are met.

S

scalar type: A C type in which the variable is a single variable, not composed of other variables.

scrolling: A method of moving the contents of a window up, down, left, or right to view contents that weren't originally shown.

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.

Edit as necessary. ⟹ **SWDS:** *Software Development System.* A development tool that lets you execute and debug applications programs by using the 'C5x debugger.

state: One of four sets of actions supported by the BTT. There are four states, labeled state 0–state 3; each state may define a maximum of four actions.

state mode: A mode of BTT operation that allows you to qualify actions based on address values only or on combinations of address and data values. Each state has its own state mode. The state mode can affect the number of actions that can be defined for a state. There are two types of state mode: address-only state mode and address-and-data state mode.

symbol table: A file that contains the names of all variables and functions in your '370 program.

system shell: A utility invoked with the SYSTEM command, which makes it possible for the debugger to blank the debugger display and temporarily exit to the DOS prompt. This allows you to enter DOS commands *or* allows the debugger to display information resulting from a DOS command.

T

target system: A '370 board that works with the emulator. Usually, the target system is a board that you have designed; you use the emulator and debugger to help you debug your design.

time-out timer: A global BTT setting that defines a time limit for running your program.

timer 1/timer 2: Timer values reported in the INSPECT window. These timers are associated with the point and range timer actions; the timer action that you define first is reported as timer 1, and the timer action that you define second is defined as timer 2.

TMS370 family: A 2–20 MHz, 8-bit, CMOS microcontroller with on-chip EEPROM storage and peripheral support functions.

TMS370Cx1x: An 8-bit, single-chip microcomputer that contains a 16-bit timer, flexible I/O, a serial peripheral interface, static RAM, and an optional data EEPROM.

TMS370Cx3x: An 8-bit, single-chip microcomputer that contains an A/D converter, flexible I/O, static RAM, optional data EEPROM, and a programmable timing module with a built-in watchdog timer.

TMS370Cx5x: A device that has the same basic features as the TMS370Cx1x with the addition of another 16-bit timer, a serial communications interface, memory expansion ports, and an 8-bit, 8-channel, A/D converter.

trace buffer: A storage area for trace samples. The trace buffer is a circular buffer that can hold a maximum of 2047 trace samples. Under certain conditions, if more trace samples are qualified, they will overwrite samples that have already been collected.

trace mode: A BTT mode that defines whether or not cycles associated with a trace sample can also be stored in the trace buffer. There are two types of trace mode: normal trace mode and TRIX mode.

trace sample: A set of information about the processor status, including values on the address and data buses, cycle information, timing information, and (when appropriate) the reverse assembly of associated code.

tracing: An action performed by the BTT when predefined conditions are met; the BTT stores a trace sample to the trace buffer.

TRIX mode: A type of trace mode in which any reads and writes associated with a qualified instruction-acquisition cycle will also be stored in the trace buffer.

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

F

T

X

Z