MWX-ICE Debugger User's Manual

SuperTAP™ Emulator for Motorola® MPC8XX

Windows Version

P/N 924-00101-00

Applied Microsystems Corporation

Applied
Microsystems
Corporation

MWX-ICE Debugger User's Manual

SuperTAP™ Emulator for Motorola® MPC8XX

Windows Version

P/N 924-00101-00

February 1997

Trademarks

Contents

Preface

Chapter 1
Introducing MWX-ICE

Chapter 2
Getting Started

Chapter 3
Using Overlay Memory

Chapter 4
Programming Flash Memory

Chapter 5
Tracing Program Execution

Chapter 6
Using Basic Breakpoints

Chapter 7
Using the Event System

Chapter 8
Support for MPC8XX Registers

Chapter 9
MWX-ICE Command Quick Reference

Chapter 10
MWX-ICE Tutorial

Appendix A
Modifying the Startup Files

Appendix B
Troubleshooting

Appendix C
Cdemon Demonstration Program

Appendix D
Updating the SuperTAP Flash ROM

Index

Preface

The MWX-ICE debugger is an integrated debugger for use with the Applied Microsystems SuperTAP™ 8XX systems integration tool.

Note

The manuals for this product are revised only at major releases of hardware or software. Changes that occur between major releases are noted only in the Help, readme, or release notes. Therefore, any differences between the Help and this document are because the Help is more current.

Documentation overview

This manual provides information that is specific to using MWX-ICE with the SuperTAP emulator for the Motorola MPC8XX processors. MWX-ICE is based on XRAY for Windows from Microtec. While MWX-ICE does not support every XRAY feature, it includes many additional features that support debugging with Applied Microsystems in-circuit emulators. For information on compatibility, see the readme file provided with this release.

For information on installing the SuperTAP hardware, see the *Emulator Installation Guide.*

MWX-ICE User's Manual

For information on	See
Conventions, support services.	This chapter
Key emulation features.	Chapter 1
Installing and starting MWX-ICE, and SuperTAP operational characteristics.	Chapter 2
Using overlay memory.	Chapter 3
Programming flash memory.	Chapter 4
Using trace capture and display.	Chapter 5
Using standard breakpoints.	Chapter 6
Using the event system.	Chapter 7
Register support for MPC8XX family.	Chapter 8
Command syntax and groupings, unsupported XRAY commands, using Help for detailed command descriptions.	Chapter 9

For information on	See
A tutorial and examples of how to use MWX-ICE features.	Chapter 10
Modifying the startup files.	Appendix A
Common startup problems and error messages.	Appendix B
How the cdemon tutorial code works.	Appendix C
Updating the emulator's firmware.	Appendix D
Finding information in this manual.	Index

Using Help

The Help for MWX-ICE also covers all the emulator-specific and core debugger features:

- Detailed command descriptions.
- Using windows, menus, and notebooks.
- Using MWX-ICE features (for example, breakpoints, event system, overlay memory, trace memory, symbol management, macros).

To get Help

From the Help menu, choose Contents, or choose Help on the currently active window. You can also search the Help file for keywords or command names. To get help on MWX-ICE debugger notebooks, click the question mark button (?) in the notebook.

For help on using Windows Help, press F1 in the Help window, or choose How to Use Help from the Help menu.

Installation and emulator setup

For information on	See
Installing the debugger	Chapter 2 of this manual.
Starting the debugger	Chapter 2 of this manual
Setting up the emulator	*SuperTAP Emulator Installation Guide*

Conventions

This manual uses the following conventions:

When you see	This means
bold type	The name of a control software configuration or executable file, a keyword or command.
italics	A command variable or a file name that you need to type. Italics are also used for emphasis the first time a key word or concept is introduced.
<F7>	Press the F7 function key.
[ ]	Optional item. You do not have to select the option. You do not type the brackets.
|	A choice between two or more options. Do not type the vertical bar.
{ }	The curly braces indicate that must choose one item. Do not type the braces.
...	You may select one or more of the items. Do not type the ellipsis.
`run`	Screen output or example code.

Support services

Applied Microsystems provides a full range of support services. New software is covered by a 90-day warranty. Support agreements are available that provide additional services.

If you encounter trouble installing or using your software, consult your manuals to verify that you are using appropriate procedures. For answers to common troubleshooting problems, see Appendix B. It covers the most frequently encountered operational problems.

If the problem persists, call Customer Support.

When you contact Customer Support, please have the ASI (Applied System Identifier) number of your system. The ASI number is printed on a label located on the bottom of the SuperTAP.

Phone

(800) ASK-4AMC (275-4262)
(206) 882-2000 (in Washington State and Canada)

See inside back page for addresses and phone numbers of worldwide offices.

Internet address

If you have access to the Internet, you can contact Applied Microsystems Customer Support using the following email address:

support@amc.com

You can also browse the Applied Microsystems World Wide Web page using the following URL:

http://www.amc.com

FAX

(206) 883-3049

Chapter 1
Introducing MWX-ICE

This chapter covers key features of the Applied Microsystems MWX-ICE debugger for the Motorola MPC8XX processors. The MWX-ICE debugger is used in conjunction with the Applied Microsystems SuperTAP system integration tool.

The following sections provide a quick overview of key features and characteristics of the SuperTAP emulator and MWX-ICE debugger. It includes pointers to additional information, required setup, and procedures.

The SuperTAP system integration tool

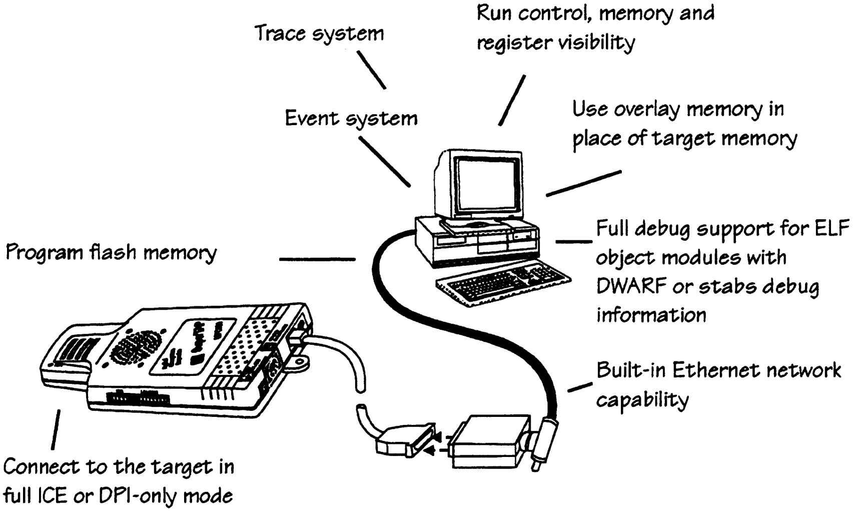
The SuperTAP is your one complete tool for system debugging and integration. You can use the SuperTAP and MWX-ICE in full in-circuit emulation (ICE) mode, where the SuperTAP replaces the target processor, or in DPI-only mode, where the SuperTAP connects to the DPI port on the target. In DPI-only mode you can access the built-in debug mode of the MPC8XX processor.

You can use SuperTAP and the MWX-ICE in full in-circuit emulation mode to bring up your target hardware, program flash memory, or to execute code from the emulator's overlay memory. The SuperTAP also supports external bus masters in multi-processor target systems.

SuperTAP provides the following features:

- Run control, memory and register visibility.
- Trace system to record processor activity.
- Event system to track and isolate deeply nested bugs.
- Overlay memory to use in place of target memory.
- Program flash memory.
- Built-in Ethernet Communications.

These features are described on the following pages.


Trace system
Run control, memory and register visibility
Event system
Use overlay memory in place of target memory
Program flash memory
Full debug support for ELF object modules with DWARF or stabs debug information
Built-in Ethernet network capability
Connect to the target in full ICE or DPI-only mode

Run control, memory and register support

Using the SuperTAP and MWX-ICE you can control target execution, examine and modify memory and registers.

Run control

You can use the run control system to single-step through your application code at the source or assembly-level. You can step into or over function calls. You can set instruction or access breakpoints in target RAM or ROM.

Basic breakpoints

Basic breakpoints are tools for interrupting emulation or simulation in order to inspect trace for insight into code execution and target function. Breakpoints interrupt emulation after memory accesses or before executing an instruction.

You use a breakpoint to examine behavior of the target under certain controlled conditions. This is very helpful in isolating bugs when troubleshooting hardware and software in the target environment.

Typically, they take two forms: access breakpoints and instruction breakpoints. You can use up to 10 single-address access breakpoints or five ranges, or some combination of both types. You can set up to 50 software instruction breakpoints.

See the **breakaccess**, **breakinstruction**, **breakread**, and **breakwrite** command descriptions in the Help. Chapter 6 provides a detailed explanation of the basic breakpoint system. The tutorial in Chapter 10 offers some practical examples of procedures and applications.

Operations during run

In the standard operating mode, MWX-ICE does not permit additional operations while the emulator is running. MWX-ICE for the MPC8XX family has a special *dynamic run mode*.

The **drun** (dynamic run) command executes the target program and continues execution until it is stopped by the **dstop** command, a breakpoint, or an error. The purpose of this mode is to allow you to interact with the emulator and debugger dynamically, while the emulator is running.

In dynamic run mode you can examine and qualify trace, set and change events and breakpoints, examine and change memory and perform most other interactive emulation functions.

You can use the **dupdate** command to poll the emulator periodically and update the windows during dynamic run mode. However, dynamic commands are no longer accepted.

The **drun, dstop, dupdate** commands provide this additional functionality. Help describes each command in detail. The tutorial in Chapter 10 includes a simple application.

Servicing interrupts during pause

Normally, a breakpoint causes the emulator to stop running in target. Using the **sitstate** command, you can enable Stop-in-Target (SIT) mode and have the emulator loop in target and service interrupts while paused, or jump to your own interrupt service routine. When **sitstate** is set for **ice** (in-circuit emulator mode), the **sit** command specifies the beginning of a minimum 4-byte block of memory at which it installs the loop routine. A branch instruction is placed at the loop address. When an interrupt request is detected, the processor services the interrupt, then returns to the branch loop.

When **sitstate** is set to **user**, the **sit** command specifies the beginning of your own interrupt service routine.

The default address for the emulator's loop or your own interrupt service routine is set by the **sit** command. This setting must be changed, either from the command line or in the Execution dialog of the Emulator Configuration window, before setting **sitstate** to **ice** or **user**.

Help provides complete descriptions of the **sit** and **sitstate** commands and explains setup.

Overlay memory

You can configure the SuperTAP with up to 8 MB of overlay memory. Overlay is RAM that can be mapped into the target system's memory space, either in place of memory or in addition to target system memory. Overlay is useful for replacing target ROM for debugging purposes, or as a stable environment when target memory is unreliable, or to temporarily expand target memory for test routines.

In addition, you can configure the SuperTAP to allow external bus masters to access overlay memory.

The **map**, **copy**, and **overlay** commands are the primary MWX-ICE commands used to map overlay and copy contents to and from target and overlay. These are described in Help. A description of overlay features, operation, and mapping is provided in Chapter 3.

Register support

MWX-ICE supports all registers of the MPC8XX family processors. Register values can be viewed, modified, and used within the conditional event system.

The SuperTAP provides:

- ❑ Windowed display and modification of all MPC8XX registers.
- ❑ Current value and descriptions of each bit in the register sets using the CPU Browser.
- ❑ Monitoring and manipulation of MPC8XX family registers during run using the event system.

Chapter 8 describes the features provided for viewing and modifying the MPC8XX registers, and lists all supported registers with the mnemonics used by the debugger.

Session logging

You can use the **log** and **journal** commands to record all session activity, including commands issued, error messages, and data returned. The **log** file can be used as an include file to replicate an earlier session. Help describes both commands.

Trace system

You can use the emulator's trace system to capture and record the execution history of the processor as the emulator executes the target program. Information captured includes CPU address, data, and status signals and timestamp information, as well as several optional fields such as interrupt activity. Using trace history, you can verify the correct performance of the software and find errors that may occur in the program's execution.

The emulator's trace buffer can store approximately 32,000 *frames* of trace. Each trace frame contains 128 bits of information. Using emulator trace capture variables and the event system, you can selectively filter the kinds of processor activity you wish to capture. Once trace has been captured, you can use trace display variables to view the contents of the trace buffer in several different ways. You can display raw bus or clock cycles, full source-level, assembly-level, or mixed source- and assembly-level trace information.

Chapter 5 explains and illustrates trace use. Help also provides complete information about trace setup, capture, and display.

Event system

The basic breakpoints feature and the event system can both be used to control emulation for insight into code execution and target function. Compared to basic breakpoints, the SuperTAP event system provides additional flexibility both in what can cause the emulator to intervene in code execution and in what actions can occur.

The emulator's event system is combined with the MPC8XX on-chip event system to provide a powerful state machine that monitors the processor bus and the event system counters, groups, and state flags. The system can track deeply nested sets of conditions, including recursive and reentrant code sections. It allows you to monitor for a predefined series of conditions, called events, and then perform emulator actions based on those conditions. It monitors target information at the bus-cycle level, including every read or write cycle that the microprocessor executes.

These features provide powerful debugging capabilities for software debugging and for hardware/software integration.

- Up to 32 when/then statements can be defined at any time.
- Four event groups and two event states provide the logical structure necessary for tracking deeply nested bugs.
- The event system includes 4 counters that can be monitored and controlled.
- Trace collection can be selectively controlled.
- Memory and register values can be monitored and modified.
- Emulation can be stopped before or after instruction execution.
- The event system can respond to or produce an external trigger signal.

Chapter 7 provides a detailed explanation of event system applications and procedures. The **when** command is the primary MWX-ICE event system command. It, and the many associated commands, are described in Help. Help also provides extensive description of event system use.

The tutorial in Chapter 10 offers some practical examples of procedures and applications.

DPI mode

You can use one tool for different kinds of debug problems. The SuperTAP can operate as a full-featured emulator and replace the processor in the target system, or it can connect to the target system using just the Development Port Interface (DPI) cable and provide simple run control, plus memory and register control. MWX-ICE automatically detects the mode of operation, and configures itself accordingly. Procedures for connecting the SuperTAP to your target system are described in the *Emulator Installation Guide*.

Flash programming

Flash memory is widely used in embedded system designs because of its non-volatility, high-performance, and low-cost. MWX-ICE and the SuperTAP provide a fast and efficient means of programming and erasing flash memory devices in your target system. The SuperTAP supports most popular flash memory devices from AMD and Intel. For information about using MWX-ICE to program flash memory, see Chapter 4 and Help.

File formats and converters

MWX-ICE requires ELF object format with DWARF debug information to enable symbolic debugging. You can also use the Gnu G++ compiler with the -gstabs+ option.

Support for other formats is built into MWX-ICE and additional converters are available.

In addition to ELF, the following formats are also recognized:

Format	Description
INTEL	Intel hex format. Extended segment address records and extended linear address records are supported.

Format	Description
SREC (default)	Motorola S3-records with Microtec extensions.
XTEK	Extended Tektronics hex format.

Symbols are not supported for these formats.

See the descriptions of **upl**, **dnl**, **uplfmt**, and **dnlfmt** in Help for supported formats, procedures, and limitations. Contact your Applied Microsystems representative for information about additional converters.

Built-in network support

With built-in Ethernet communications support, the SuperTAP is network ready. Engineers can share access to the SuperTAP and the target system without having to add additional hardware to the target. The SuperTAP supports the widely-used TCP/IP network protocol. If your network uses RARP or BOOTP servers, the SuperTAP can automatically configure its IP address and netmask using your network database. Procedures for connecting the debugger to a networked SuperTAP are described in Chapter 2 of this manual.

For information about configuring the SuperTAP hardware for Ethernet communications, see the *Emulator Installation Guide*.

Chapter 2

Getting Started

This chapter explains how to install and start the debugger and provides important information about characteristics of the emulator and debugger. Before you begin debugging, you should familiarize yourself with this information.

Installing MWX-ICE for Windows

These instructions describe how to install the Applied Microsystems MWX-ICE debugger on a PC or compatible computer that is running Microsoft Windows 95 or Windows NT version 4.0.

- □ Install the MWX-ICE software by following the instructions included with the CD-ROM.
- □ Configure the emulator for communications as described in the *Emulator Installation Guide.*
- □ Set the environment variables as described in "Setting up environment variables" on page 2-2.

System requirements for MWX-ICE

The Applied Microsystems MWX-ICE debugger runs on a PC or compatible computer that is running Microsoft Windows 95 or Windows NT version 4.0.

For information on the specific host, network, and emulator requirements for this release of MWX-ICE, see the release letter and readme shipped with your order.

MWX-ICE uses the Transmission Control Protocol/Internet Protocol (TCP/IP) to communicate with the SuperTAP. Support for TCP/IP is built into Windows NT. If you are using Windows 95, you need to add the Microsoft TCP/IP communications protocol.

Setting up environment variables

The default installation directory for MWX-ICE is C:\ST8XX. If you use this directory, you do not need to set up the XRAYMASTER environment variable, and you can skip this step.

Search order

MWX-ICE searches for STARTUP.INC and other MWX-ICE support files in the following order:

1. *current_directory*\STARTUP.INC
2. *current_directory*\AMC\ST8XX\filename
3. XRAYMASTER\AMC\ST8XX\filename
4. C:\ST8XX\AMC\ST8XX\filename

Setting XRAYMASTER

If you have installed the MWX-ICE debugger in a directory different from the default, you need to set up the XRAYMASTER environment variable in your AUTOEXEC.BAT file.

The syntax is:

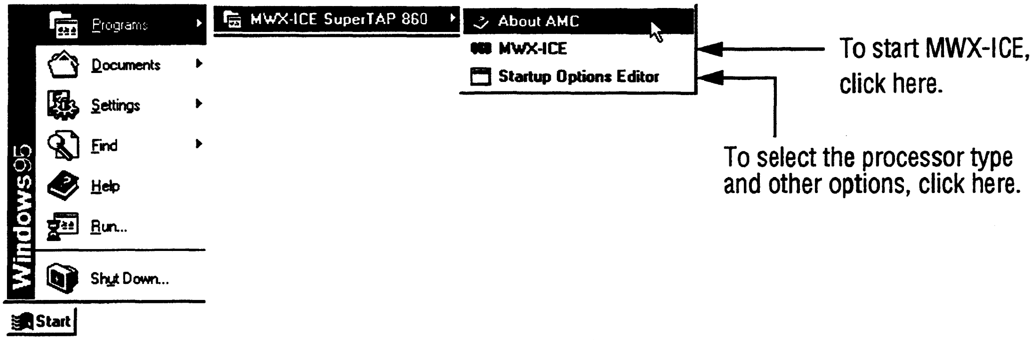
```
set XRAYMASTER=install_dir
```

Starting MWX-ICE

To start MWX-ICE, you must have met the following requirements:

- ❑ MWX-ICE must be installed, and environment variables must be set.
- ❑ The SuperTAP must be configured for Ethernet communications as described in the *SuperTAP Emulator Installation Guide.*
- ❑ A TCP/IP protocol stack must be running on your host computer, and your network must have some means of performing address resolution.

This following section summarizes startup procedures for MWX-ICE.



To start MWX-ICE, click here.

To select the processor type and other options, click here.

Figure 2-1 MWX-ICE group in the Windows Start menu

Using the Startup Options Editor

To make selecting startup options easier, you can use the MWX-ICE Startup Options Editor. The editor creates a startup options file (MWX.CFG) that is automatically included when you start the debugger.

➤ To use the Startup Options Editor

1. Click the Start button, and point to Applied Microsystems, then point to MWX-ICE SuperTAP 8XX, and click the Startup Options Editor.

   The Startup Options Editor dialog appears.

2. Select the processor variant you want to emulate.

   You must change this selection each time you change the processor you are emulating.

3. Select other options as needed.

   For information on the available options, choose Help.

4. Choose OK.

The options editor saves your choices to a file (the default is MWX.CFG).

➤ To start MWX-ICE

- Click the Start button, and point to Applied Microsystems, then point to MWX-ICE SuperTAP 8XX, then click MWX-ICE.

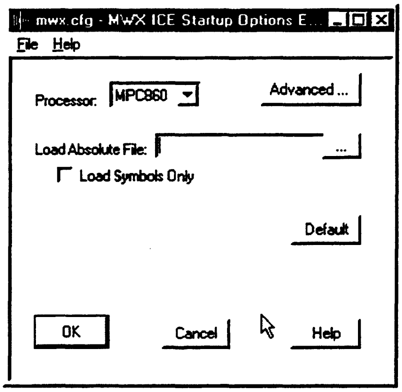
  The debugger automatically looks for the file MWX.CFG, and uses those options at startup. If you save the options to a different file, you need to add a new program icon to the Start or Programs menu. For information, click Help from the Startup Options Editor, and see the "Add a new program" topic.

  If you want to use the same startup options the next time you start MWX-ICE, just double-click the appropriate MWX-ICE icon.

  The first time you start up MWX-ICE, the debugger comes up in the unconnected state. For information on defining a connection, see "Connecting to an emulator" on page 2-7.

Note

For more information about the startup options file, see “About the startup options file (MWX.CFG)” on page A-7.



mwx.cfg - MWX ICE Startup Options E...

Processor: MPC860

Load Absolute File:

☐ Load Symbols Only

Figure 2-2 MWX-ICE Startup Options Editor

Connecting to an emulator

When you start up MWX-ICE you won't automatically connect to an emulator unless you have saved your configuration to a startup file. If MWX-ICE is unable to locate a startup file (STARTUP.INC) the Connections window appears along with the Code and Command windows. You can use the Connections window to define and connect to an emulator. Once you have connected to an emulator, you can save the configuration.

The section describes the steps needed to connect to an emulator. Before you can connect to an emulator, you must first configure the emulator for Ethernet communications as described in the *Emulator Installation Guide*.

Starting the emulator

The SuperTAP has special buffering to protect the emulation circuitry. To function, emulator power must be on. Use the following sequence when powering on the emulator and target.

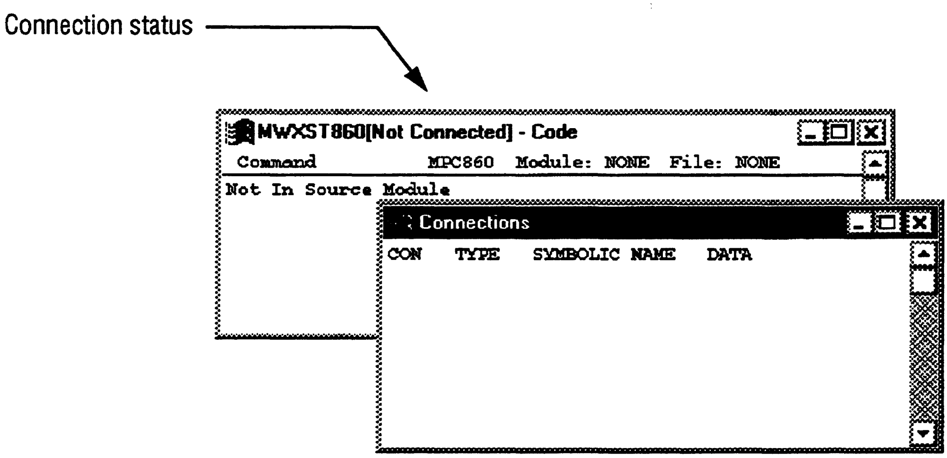
If you do not have a target, the emulator goes into isolation mode. For information about using isolation mode, see "Isolation mode" on page 2-24.

Note

The first time you start MWX-ICE, do not have the target connected. This will allow you to come up in isolation mode and configure MWX-ICE for your target requirements. Save the configuration so that MWX-ICE will be configured correctly at startup when your target is connected.

➤ To start the emulator

1. Turn on power to the emulator.
2. Turn on power to the target.



```
MWXST860[Not Connected] - Code
 Command            MPC860  Module: NONE  File: NONE
Not In Source Module

Connections
CON    TYPE    SYMBOLIC NAME    DATA
```

Figure 2-3 Connections window [Not Connected]

Defining a connection

Before you can connect to an emulator, you must define an emulator connection. You can define as many connections as you like, but you can only connect to one emulator at a time.

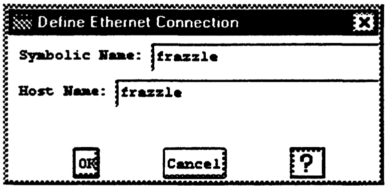
➤ To define a connection

1. From the Displays menu, select Connections.

   The Connections window appears. The Connections window automatically appears if MWX-ICE is unable to locate a startup include file.

2. From the Actions menu, choose Define Ethernet Connection.

   The Define dialog box appears.



Symbolic Name: frazzle

Host Name: frazzle

3. In the Host Name box, type the name of the emulator as it is known on your network.

4. In the Symbolic Name box, type a name you can use to identify the emulator.

   Note that the Symbolic Name is used by the **connect** command. The Symbolic Name provides easy way to label the different communications configurations. For example, for Ethernet communications, you can use the host name of the emulator as the Symbolic Name.

5. Click OK.

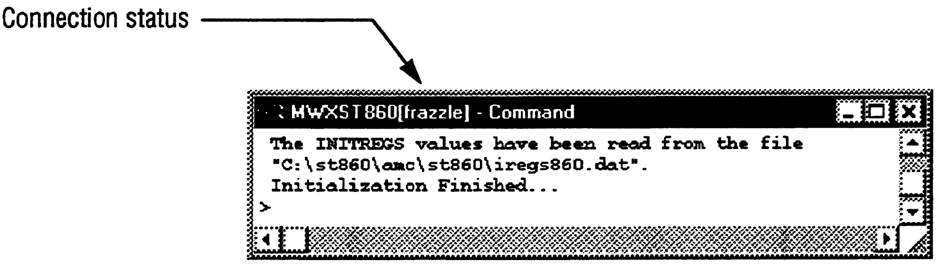
Making a connection

The emulator you connect to must support the connection type you have selected. Be sure the emulator is on and is configured properly. If you disconnect from one emulator during a debug session and connect to another, the emulators must have the same type of processor, the processor you selected using the Startup Options Editor.

➤ To connect to an emulator

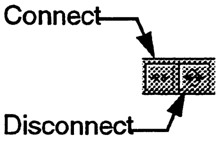
1. In the Connections window, double-click to select an emulator.
2. Click the Connect button.

   An asterisk (*) appears in the CON column when you are connected to the emulator. The name of the connection appears in the title bar of the Command window. Status information appears in the Command window.



```
The INITREGS values have been read from the file
"C:\st860\amc\st860\iregs860.dat".
Initialization Finished...
>
```

Figure 2-4 Command window showing connection [frazzle]



TIP: You can drag and drop the Connect button from the tool bar on an emulator in the Connections window.

Configuring the emulator and debugger

Once you've connected to an emulator, you can configure the debugger and emulator options. For a brief description of the options you can set, see "Configuring the emulator" on page 2-12.

Saving a connection

Once you have defined and connected to an emulator, you can save the connection and configuration to a startup include file. When you have a startup include file (STARTUP.INC), MWX-ICE automatically connects to the emulator and configures the emulator options you have set.

➤ To save a configuration to STARTUP.INC

1. From the Displays menu, choose Emulator Configuration.
2. In the Emulator Configuration window, choose Save to Startup from the File menu.

   By default, your connection and configuration data are saved to C:\ST8XX\STARTUP.INC.
3. If the file exists, a prompt appears asking if you want to overwrite the existing file, or replace the configuration section of the existing file.
   - Choose replace if you have added commands to an existing startup file. (See "Adding commands to the startup file" on page A-5 for procedures to add non-configuration commands to a startup file.)
   - Choose overwrite if you don't want to keep the existing file or any commands you've added.

Note

The Save to Startup command saves the configuration to whatever file you specified at start up. If no file was specified, MWX-ICE uses the default STARTUP.INC file. If this file doesn't exist, MWX-ICE asks if you want to create it.

➤ To save a configuration to another file

1. Switch to the Emulator Configuration window.
2. From the File menu, choose Save to File.
3. Specify the path and filename.
4. Save your configuration to a new startup file.

Configuring the emulator

Once you've connected to an emulator, you can configure the debugger and emulator options. To set the options, open the Emulator Configuration window and click the button for the group of options you wish to set.

The configuration dialogs show the current settings for the options. Use the menus or text boxes to change the settings, and then choose the Apply button. For information about the options in the configuration dialogs, click Help.

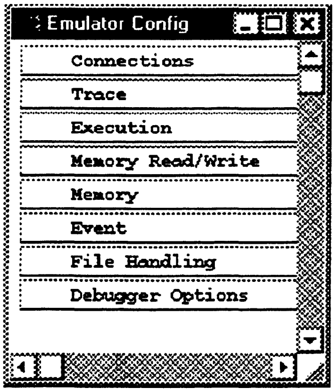


Figure 2-5 Emulator Configuration window

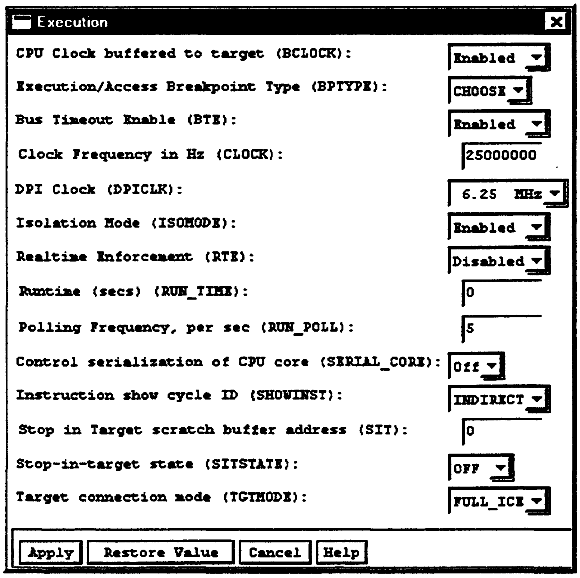
You can use the Emulator Configuration window to view and modify the options that control the state of the debugger and emulator.

Connections

The Connections button brings up the Connections window. Use this window to define and connect to emulators, and to reload the emulator operating system. The Connections window is shown on page 2-10.

Execution

The Execution button opens the Execution configuration dialog box. Use this dialog box to set the emulator execution options, such as instruction show cycles, isolation mode, and real-time operation.



CPU Clock buffered to target (BCLOCK): Enabled

Execution/Access Breakpoint Type (BPTYPE): CHOOSE

Bus Timeout Enable (BTE): Enabled

Clock Frequency in Hz (CLOCK): 25000000

DPI Clock (DPICLK): 6.25 MHz

Isolation Mode (ISOMODE): Enabled

Realtime Enforcement (RTE): Disabled

Runtime (secs) (RUN_TIME): 0

Polling Frequency, per sec (RUN_POLL): 5

Control serialization of CPU core (SERIAL_CORE): Off

Instruction show cycle ID (SHOWINST): INDIRECT

Stop in Target scratch buffer address (SIT): 0

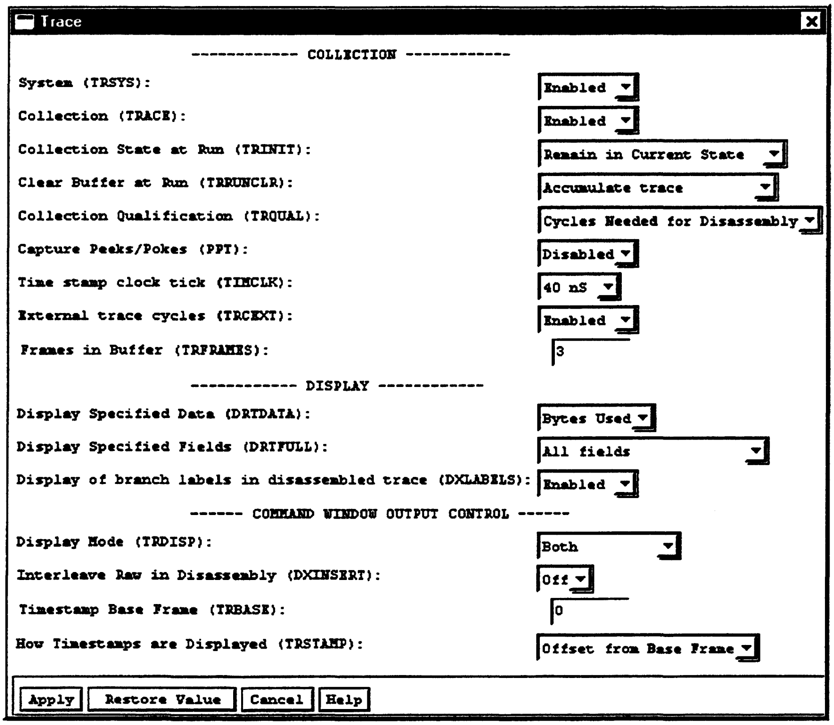
Stop-in-target state (SITSTATE): OFF

Target connection mode (TGTMODE): FULL_ICE

Figure 2-6 Execution configuration dialog box

Trace

The Trace button opens the Trace configuration dialog box. Use this dialog box to set emulator trace collection and display options. For information about capturing and viewing trace, see Chapter 5.



------------ COLLECTION ------------

System (TRSYS): Enabled

Collection (TRACE): Enabled

Collection State at Run (TRINIT): Remain in Current State

Clear Buffer at Run (TRRUNCLR): Accumulate trace

Collection Qualification (TRQUAL): Cycles Needed for Disassembly

Capture Peeks/Pokes (PPT): Disabled

Time stamp clock tick (TIMCLK): 40 nS

External trace cycles (TRCEXT): Enabled

Frames in Buffer (TRFRAMES): 3

------------ DISPLAY ------------

Display Specified Data (DRTDATA): Bytes Used

Display Specified Fields (DRTFULL): All fields

Display of branch labels in disassembled trace (DXLABELS): Enabled

------ COMMAND WINDOW OUTPUT CONTROL ------

Display Mode (TRDISP): Both

Interleave Raw in Disassembly (DXINSERT): Off

Timestamp Base Frame (TRBASE): 0

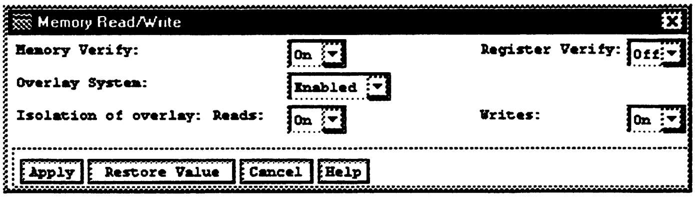
How Timestamps are Displayed (TRSTAMP): Offset from Base Frame

Figure 2-7 Trace configuration dialog box

Memory

The Memory Read/Write button opens the Memory Read/Write dialog box. Use this dialog box to enable overlay memory, and to control access to overlay by external bus masters (Isolation of overlay read/write).

The Memory button opens the Memory configuration dialog box. Use this dialog box set memory access attributes. For information about using the emulator's overlay memory, see Chapter 3.



Memory Read/Write

Memory Verify: On

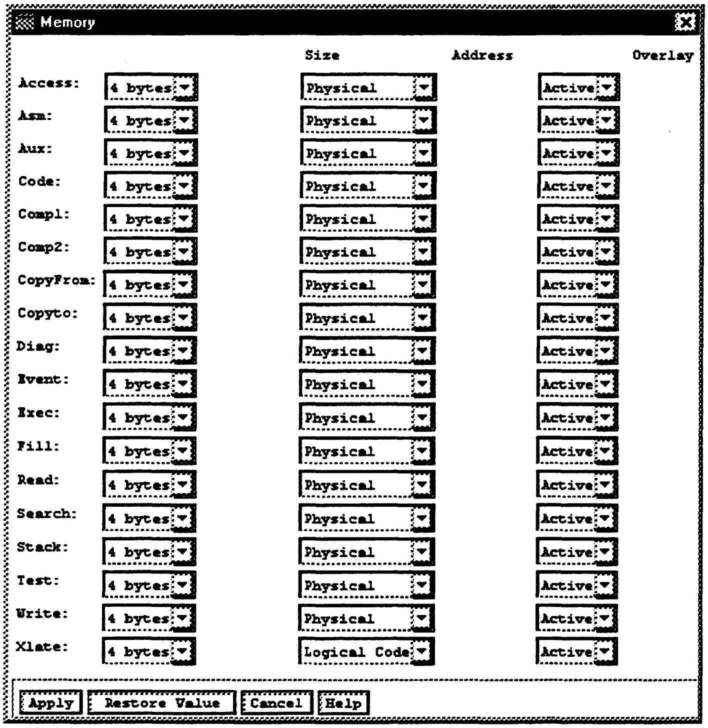
Register Verify: Off

Overlay System: Enabled

Isolation of overlay: Reads: On

Writes: On

Figure 2-8 Memory Read/Write configuration dialog box

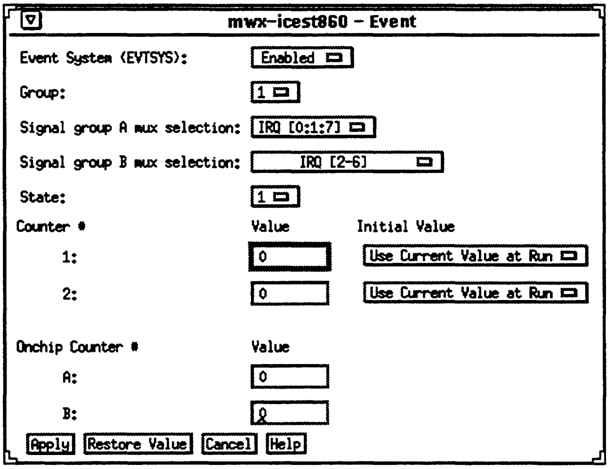


		Size	Address	Overlay
Access:	4 bytes	Physical		Active
Asm:	4 bytes	Physical		Active
Aux:	4 bytes	Physical		Active
Code:	4 bytes	Physical		Active
Comp1:	4 bytes	Physical		Active
Comp2:	4 bytes	Physical		Active
CopyFrom:	4 bytes	Physical		Active
Copyto:	4 bytes	Physical		Active
Diag:	4 bytes	Physical		Active
Event:	4 bytes	Physical		Active
Exec:	4 bytes	Physical		Active
Fill:	4 bytes	Physical		Active
Read:	4 bytes	Physical		Active
Search:	4 bytes	Physical		Active
Stack:	4 bytes	Physical		Active
Test:	4 bytes	Physical		Active
Write:	4 bytes	Physical		Active
Xlate:	4 bytes	Logical Code		Active

Figure 2-9 Memory configuration dialog box

Event

The Event button opens the Event configuration dialog box. Use this dialog box to set the emulator event system options. For information about using the event system, see Chapter 7.



mwx-icest860 - Event

Event System (EVTSYS): Enabled

Group: 1

Signal group A mux selection: IRQ [0:1:7]

Signal group B mux selection: IRQ [2-6]

State: 1

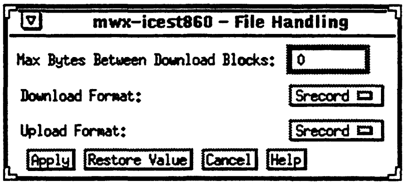
Counter #	Value	Initial Value
1:	0	Use Current Value at Run
2:	0	Use Current Value at Run

Onchip Counter #	Value
A:	0
B:	0

Figure 2-10 Event configuration dialog box

File Handling

The File Handling button opens the File Handling configuration dialog box. Use this dialog box to specify the upload and download format for non-ELF object files, and to other download options.



mwx-icest860 - File Handling

Max Bytes Between Download Blocks: 0

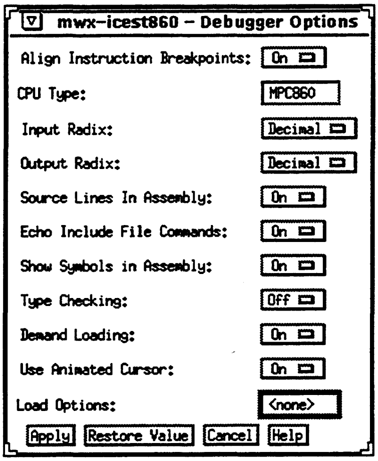
Download Format: Srecord

Upload Format: Srecord

Figure 2-11 File Handling dialog box

Debugger Options

The Debugger Options button opens the Debugger options configuration dialog box. Use this dialog box to set input and output radix and other debugger options.



mwx-icest860 - Debugger Options

Align Instruction Breakpoints: On

CPU Type: MPC860

Input Radix: Decimal

Output Radix: Decimal

Source Lines In Assembly: On

Echo Include File Commands: On

Show Symbols in Assembly: On

Type Checking: Off

Demand Loading: On

Use Animated Cursor: On

Load Options: <none>

Figure 2-12 Debugger Options dialog box

Other things to do at startup

- ❑ Map any overlay memory needed. See Chapter 3 or Help for procedures.
- ❑ Download target code to overlay, if needed. See Chapter 3. You can use the Startup Options Editor to enter the name of the absolute file to load.
- ❑ Once you've started the emulator and debugger, you need to initialize the processor using the **reset** command, then you can begin emulation.
- ❑ Set up the *initialization registers*. These registers provide a way of decoding the multiplexed control pins, disabling the software watchdog timer, and making MWX-ICE memory operations possible immediately after a processor reset. See "Using the set of initialization registers" on page 8-2.

Isolation mode

The debugger provides two operational modes: in-target and isolation mode. If you have no target to connect to or the target is unreliable, you can run in isolation mode using the probetip processor and downloading target code to overlay memory (if installed). The system automatically enters isolation mode at startup if no target V_{CC} is found; so no special setup is required.

Note

Special procedures do apply if target V_{CC} is lost and isolation mode is entered after startup. See “Support for target power loss” on page 2-24.

You can enter isolation mode by enabling the isolation mode option in the Execution Configuration dialog. To enable this mode, open the Execution Configuration dialog from the Emulator Configuration window. You can also use the **isomode** softswitch in the Command window.

The **isomode** softswitch controls the operational mode. The internal default is in-target mode (**isomode off**). Use of the **isomode** softswitch is described in Help.

If you use isolation mode, you need to consider the following:

- Whenever isolation mode is enabled, the emulator does not use the target, even if one is connected. If the emulator fails to find a connected target, check the setting of the Isolation mode option in the Execution Configuration dialog. Make sure that it is disabled, and then save your configuration to the STARTUP.INC file. If the target still is not found, or if the emulator shifts to isolation mode, check target power.
- When connected to a target with **isomode** on, all signals between the target and probetip float. If your target cannot tolerate floating lines, disconnect it from the probetip when isomode is on, or always keep **isomode** off.
- During times when you wish to emulate and debug in a targetless environment, it is a good idea to enable the Isolation Mode option, and then save your configuration to the STARTUP.INC file.

What happens at power on, reset, and restart

When you turn power on to the SuperTAP emulator and start the MWX-ICE debugger, two parallel processes begin. On the emulator side, the controller runs PROM-based diagnostic and boot code. During this process, the hardware installed in the emulator is polled and its hardware configuration stored for future reference.

On the host side, the Applied Microsystems debugger queries the SuperTAP for its state and configuration. If the emulator reports itself in a power-up state, the program checks for the presence of the emulator control shell and downloads it, if necessary.

When the control software completes its configuration and is running, the emulator's state changes to "Ready," and the control software is ready to run.

The following describes each portion of the process in detail.

Emulator power-on sequence

The following sequence of events occurs when you turn on power to your SuperTAP emulator:

- ❑ The emulator controller runs PROM-based diagnostic and boot code.
- ❑ The controller board is initialized.
- ❑ The configuration stored in the SuperTAP's flash memory is read.
- ❑ The emulator starts the emulator control software, if it is loaded into flash memory.
- ❑ The emulator is reset to a known state.
- ❑ The breakpoint, event, trace, and overlay systems are cleared and initialized.

Emulator control program startup

When you start MWX-ICE, the following events occur:

- The host opens the startup.inc file (if it exists) and loads the connection and configuration parameters.
- The host screen and search paths are initialized, the emulcfg.dat database is opened, and the host-emulator link is initialized.
- The emulator interface layers are initialized and opened.
- The emulator is reset to a known state.
- The breakpoint, event, trace, and overlay systems are cleared and initialized.

Debugger reset

The **reset** command allows you to regain control of your target and re-synch the emulator with it. It is required to recover from timeout errors.

What happens when you use the reset command

When you execute **reset** from the command line, the emulator performs the following:

- Asserts $\overline{\text{HRESET}}$ to reset the processor on the SuperTAP.
- Restores the SuperTAP logic to a known state.
- Preserves memory mappings and other emulation settings.
- Processes the initialization registers (if **initregs** is enabled).

The **reset** command does not re-initialize memory; variables are not reset to original values. Use the **load** or **reload** commands to restore variables.

If this command is issued while the emulator is in **drun** mode, emulation stops before execution of the command.

Program restart

If the symbol table is loaded, **restart** resets the program counter and stack pointer to the original starting address from the absolute file. The next time you go into run, execution restarts at the beginning of the program. Breakpoints are not cleared, variables are not reset, and any declared I/O ports still exist as originally specified.

Emulator reset

In rare situations, you may have to reset the SuperTAP to restore the system to a known state.

➤ **To reset the SuperTAP**

1. Exit the debugger.
2. Push the toggle switch on the back of the emulator to the reset position.
3. Restart the debugger.

What happens when you press reset

When you press the reset switch, or cycle power, the following sequence of events occurs in the emulator:

- ❑ Startup diagnostics are performed.
- ❑ Communication parameters are loaded from the system core, and the appropriate Ethernet protocol is used to establish communications across the network.
- ❑ The SuperTAP's flash memory is checked for the presence of a transaction shell.

  If no shell is found, the transaction shell is downloaded from the host to the SuperTAP.

  If a shell is found, it is started, the target processor is reset, and the emulator enters pause mode and waits for connection with the debugger.

During the reset process, the SuperTAP enters isolation mode if the emulator is not connected to a target, or if no target power is detected.

During the reset process, DPI-only or full emulation mode is selected based on whether power comes from the DPI cable, or the target. From MWX-ICE, the **tgtmode** command shows which mode is selected.

Software watchdog timer

The MPC8XX has a software watchdog timer (SWT) that is enabled after a system reset to cause a system reset when it times out. If you don't plan to use the SWT, you must clear the software watchdog enable bit (SWE) in the system protection control register (SYPCR) to disable the timer.

Because the SWT is enabled after a reset, you can use the SuperTAP's initialization registers (**initregs**), to automatically disable (or configure) the timer after reset. This way you won't have to run your boot code or manually configure register after each reset operation. If you are not using **initregs**, the SWT times out every four seconds and resets the processor.

To avoid this problem, you must enable the initialization registers (**initregs on**), and save then initialization registers to a file.

For information about setting up the initialization registers, see "Using the set of initialization registers" on page 8-2.

Important operational notes

This section presents various characteristics of the emulator and debugger that you should be aware of during emulation. It includes:

Support for target power loss

Target power is monitored. Anytime an operation is attempted when VDDH, VDDL, or KAPWR not present, an error is generated.

With the emulator installed, only approximately 1 mA is drawn from target VCC through the CPU socket. Note that this is substantially less than the current drawn by an actual CPU.

Isolation mode

An emulator softswitch (**isomode**) enables selection of an internal clock when target power is lost.

Caution ⚠

When the SuperTAP is connected to a target and **isomode** is on, the bus, clock, and power signals are isolated, and all CPM signals are connected to the target. You should always disconnect the SuperTAP from the target when **isomode** is on, or always keep **isomode** off.

Search

Keywords:
ISOMODE
Vcc

Loss of power during run

If target power is lost while the emulator is running, MWX-ICE issues a warning message. Whether power remains off or is restored, the emulator remains in run and maintains emulation unless the Stop button is clicked. Stopping while power is lost may cause corruption of code, even in overlay, because breapoints can't be removed.

Loss of power while in pause

If target power is lost while the emulator is paused, MWX-ICE issues a warning message when you next enter a debugger command.

A target reset is forced whenever target power is restored.

Peek and pokes during pause or run

Peeks are reads, and pokes are writes performed during pause. These may occur as a result of event system activity (reading a register, incrementing a variable, etc.) or when you look at or modify memory while emulation is paused. The tracing of peek/poke cycles is controlled by the **ppt** command.

Various display characteristics

Several characteristics of MWX-ICE windowing and display could cause confusion unless understood:

- During high-level single-stepping. the highlight bar seems to jump randomly from line to line in the Code window. This is correct behavior usually resulting from highly optimized code.
- Error dialog boxes are scrollable. Occasionally, it is necessary to scroll up or over to view the entire message.

Show cycles

The SuperTAP is designed to operate even when you are using the processor instruction and data caches. The SuperTAP trace system requires that the processor instruction show cycles are enabled. In most cases, you only need to generate show cycles for indirect branching. The MWX-ICE **showinst** command controls processor show cycles. The default setting is for **showinst** is indirect. Enabling show cycles for indirect branching only adds a minor performance penalty to the processor execution. But this is hundreds of times better than the performance hit you take for disabling caches. The **serial_core** command controls processor serialization. In most cases, you won't need to serialize the core to be able to trace execution or use the event system. For more information on the **showinst** and **serial_core** commands see Help.

AC timing

The SuperTAP AC signal timing is exceptionally close to the emulated processor. In most cases, it adds only 1 to 2 nanoseconds to published Motorola timing for external signals and should operate well within Motorola worst-case timing. It can add 1.8 to 3.0 nanoseconds to the CLKOUT signal, depending upon whether you need to add additional buffering.

Exhaustive measurements to verify calculated worst case numbers have not been made. All calculated values assume worst-case timing at 40 °C ambient. Target signals are expected to be properly terminated to avoid reflections. For more information about AC timing, see the *SuperTAP Emulator Installation Guide*.

AC timing specifications are provided by Applied Microsystems as general guidelines for customers using our products. These specifications are calculated and measured under the conditions specified. Because of variations in target loading, temperature and device timings, Applied Microsystems does not guarantee these specifications. Applied Microsystems reserves the right to make changes to these specifications at any time without notice.

Alternate bus master

The emulation logic is designed so that the monitored address, data, and status signals remain valid even when the processor has lost the bus to an alternate bus master. This allows the trace and event systems to continue to operate during alternate bus master cycles. To enable tracing of external bus cycles, configure the emulator's **trcext** variable to on.

Recoverable interrupts

The MPC8XX has a bit in the machine state register (MSR) called the recoverable interrupt bit (MSR_{RI}). The MSR_{RI} indicates whether the interrupt is restartable. If this bit is not set, the target CPU may not respond to breakpoints. To the processor, a normal, maskable break looks just like any other interrupt/exception.

For example, if you want to set a breakpoint at the beginning of an interrupt service routine, you need to ensure that the recoverable interrupts are enabled, and that the machine status save/restore registers (SSR0/SSR1) are correctly written.

To handle exceptions, your interrupt service routine must do the following:

1. Save the SSR0 and SSR1 registers to memory.
2. Set the $\mathrm{MSR}_{\mathrm{RI}}$ bit.
3. Execute any exception processing.
4. Clear the $\mathrm{MSR}_{\mathrm{RI}}$ bit.
5. Restore the SSR0 and SSR1 registers.
6. Execute the rfi system call.

For more information on recoverable interrupts, see the *Motorola MPC8XX User's Manual.*

Support for the MMU and logical addressing

The SuperTAP provides support for the memory management unit of the MPC8XX. Prior to any emulator action, a valid translation table must be loaded into memory before the instruction or data MMU is enabled.

All logically addressed user input (breakpoints, overlay maps, etc.) is translated to the corresponding physical addresses before being applied. During run, the system uses the translation tables to perform logical-to-physical translation dynamically.

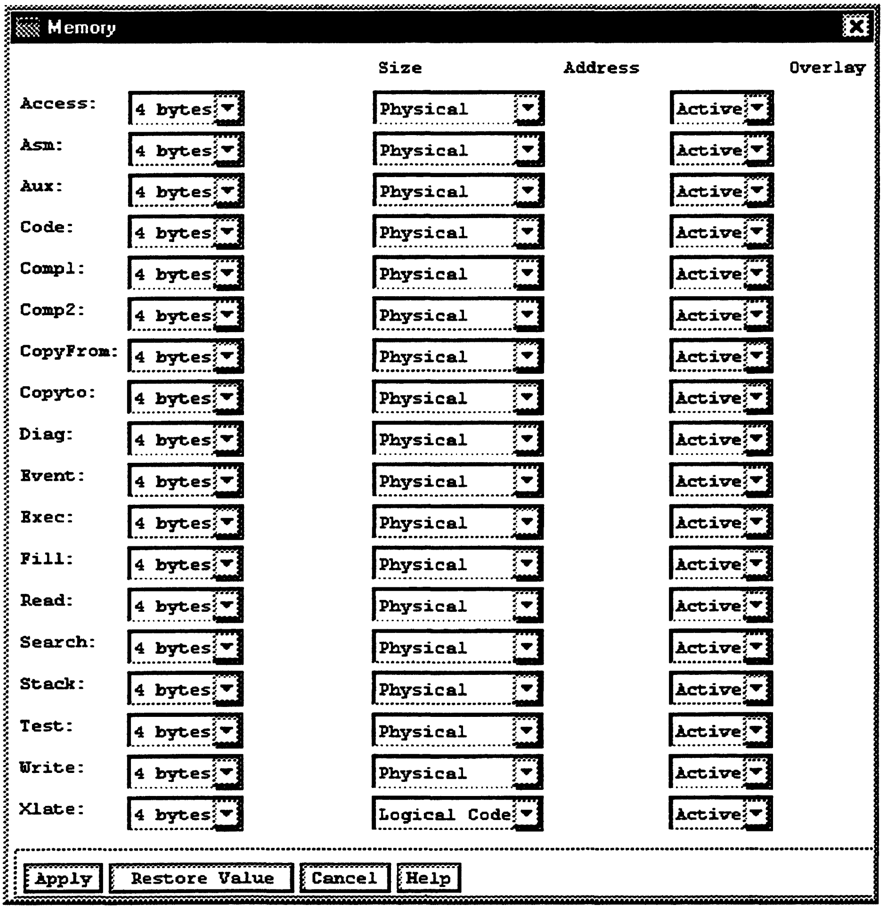
Address translation

Every function of the emulator and debugger supports logical code, logical data, and physical addressing. As long as valid translation tables are loaded into memory, the SuperTAP emulator can automatically translate logical addresses to physical. The **xlate** utility provides the means for manual logical-to-physical translations.

To indicate whether the default addressing mode is logical or physical for the many key functions of the debugger, use the Memory Configuration dialog box (see Figure 2-1), or the matrix provided by the **address** command. Each row identifies a type of memory activity performed using a specific command or feature of the debugger. The setting for each specifies how the debugger should interpret subsequent address input it receives. If incorrectly set to logical, these settings may cause commands or features affected by them to function unpredictably or cause exceptions.

In some cases these defaults can be overridden individually. For example, the event system field (**event**) is applied only if no overriding qualifiers are specified in the **when** inputs. In all other cases (**ba**, **bi**, etc.), you can toggle the setting prior to individual command input to change the addressing mode for the following entry. The startup.inc file can be used to set defaults at startup.

Regardless of the mode of user input, the system uses physical addressing during operation.



	Size	Address	Overlay
Access:	4 bytes	Physical	Active
Asm:	4 bytes	Physical	Active
Aux:	4 bytes	Physical	Active
Code:	4 bytes	Physical	Active
Comp1:	4 bytes	Physical	Active
Comp2:	4 bytes	Physical	Active
CopyFrom:	4 bytes	Physical	Active
Copyto:	4 bytes	Physical	Active
Diag:	4 bytes	Physical	Active
Event:	4 bytes	Physical	Active
Exec:	4 bytes	Physical	Active
Fill:	4 bytes	Physical	Active
Read:	4 bytes	Physical	Active
Search:	4 bytes	Physical	Active
Stack:	4 bytes	Physical	Active
Test:	4 bytes	Physical	Active
Write:	4 bytes	Physical	Active
Xlate:	4 bytes	Logical Code	Active

Figure 2-1 Memory configuration dialog box

Table 2-1 Address matrix of memory qualifiers

Label	Sets Default Addressing Mode for
ACCESS	Access breakpoints set using **ba**, **br**, **bw.**
ASM	Entries using the **asm** line assembler.
AUX	SIT loop location.
CODE	Downloads (**load**, **dnl**) and code display refreshes.
COMP1	First argument of **compare**.
COMP1	Second argument of **compare**.
COPYFROM	Source of **copy** action.
COPYTO	Destination of **copy** action.
DIAG	Memory tested by diagnostic routines (requires physical) and **crc**.
EVENT	Any event system address entries.
EXEC	Execution breakpoints set with **bi**, high-level step and BreakI.
FILL	Memory to be modified by **fill**.
READ	Reads using **dump**, **upl**, **disassemble.**
SEARCH	Memory locations specified in **search**.
STACK	Stack display and refresh.
TEST	Memory accessed by **test**.
WRITE	Writes using **setmem**.
XLATE	Address translations using **xlate**.

Mapping logically addressed memory in overlay

When you map a logically addressed range to overlay memory, the following occurs:

- The system translates the start addresses to a physical address.
- Using the length specified in the **map** range, it maps overlay memory as a continuous block.

For this reason, the logical and physical memory must map to contiguous physical addresses. Chapter 3 explains overlay use and describes procedures and limitations when using logical addressing. Overlay addresses are always displayed as physical values.

Copying using logical addresses

One special consideration applies when you request a copy within target or overlay memory, rather than between target and overlay.

The copy function of the debugger relies on the assumption that source and destination logical memory translates to contiguous blocks of physical memory. In the unlikely event that this is not the case, there is a chance that the source range can be overwritten before all necessary reads are performed.

Once it translates the source and destination start addresses, the debugger requests new translations during the copy only at large boundaries. If the underlying physical memory for the range is not contiguous, it is possible to overwrite the source range before it has been read, and then copy this incorrect data to the destination range.

Addresses in displays

Except for disassembled trace and any logical address showing in the raw trace for the Instruction Pointer, the debugger displays addresses in physical addressing mode, regardless of the addressing mode used for address entry. Consequently, you should understand the logical-to-physical mappings for the memory you are debugging.

Raw trace This trace shows any logical address for the IP and uses physical addressing in the remaining display. This includes raw trace interleaved with disassembled trace.

Overlay map The listings of mapped memory use physical addresses.

Operational considerations

If you plan to attempt workarounds for table and paging methods not supported during run, there are several considerations to bear in mind:

- If during operation your code changes memory translation, you must structure your debugging session to deal with one translation table at a time. The emulator cannot automatically disable breakpoint, event, and other address-dependent configurations that would no longer apply when a new translation takes effect. For example, any overlay mapping must be adjusted and appropriate code loaded into memory.
- If two or more logically addressed elements share the same physical address, there is the chance of unpredictable emulation behavior. The emulator converts all logical addressing to the corresponding physical address. The possibility exists that a breakpoint or event comparator would cause a break or other action at an unintended point during operation.

Chapter 3

Using Overlay Memory

During development and integration, you often need stable memory to replace or extend actual target memory. The optional overlay features of the SuperTAP emulation system provide these capabilities.

This chapter provides information about overlay procedures, use, and related topics.

Using Help

The information presented here supplements the related overlay topics found in the Help. You may wish to have Help running while you read this chapter. Where applicable, the relevant keyword search term is included in a marginal callout Search .

How overlay is used in debugging

Overlay has four typical purposes:

- Provide a substitute for target memory during early integration when target memory is not operational.
- Offer a stable substitute into which to load code and test against target operation.
- Replace the ROM on the target, enabling you to patch on the fly without recompiling, rather than continually burning new PROMs.
- Extend target memory temporarily to hold test routines and code expansion beyond ROM and RAM limits.

Typical process

1. Set memory configuration or use the INITREGS feature.
2. Map the memory regions to overlay.
3. Copy the PROM contents or load code into overlay.
4. Run to a breakpoint or event condition.
5. Examine trace.
6. Use the line assembler to patch and retest without recompiling, even in ROM space.
7. Upload patched object code to host for re-use.

Features and important characteristics of overlay

Overlay memory is physical memory provided by the emulation system. It can be mapped into the target system's memory space and programmed to respond as if it were target system memory. This means you can replace high-speed target memory subsystems with comparable overlay during debugging.

The custom features provided by the SuperTAP emulator are summarized below.

Overlay options

The SuperTAP can be equipped with up to 8 MB of overlay memory. Overlay memory is available in 1 MB, 4 MB, or 8 MB modules. Contact Applied Microsystems if you want to increase the amount of overlay in your system.

Wait states

Overlay responds to accesses with the following speed characteristics:

- 0 wait states for 25 MHz bus speeds
- 1 wait state 50 Mhz bus speeds.

Overlay functions correctly in regions configured with 32-, 16-, or 8-bit port sizes.

Termination

When a memory bank is configured for external $\overline{\text{TA}}$ generation, (the SETA bit in the chip-select option register), the target must assert the $\overline{\text{TA}}$ signal even if the memory bank has been mapped to overlay memory. In the case of isolation mode (**isomode on**), the SuperTAP overlay system will supply one wait state $\overline{\text{TA}}$.

128K minimum granularity

Overlay memory can be mapped anywhere in logical or physical memory in ranges as small as 128K. The mapper automatically adjusts non-conforming ranges to match grain requirements and aligns ranges on 128K boundaries. This is explained in "How memory mapping is handled" on page 3-6.

Search Keywords: MAP

Qualification by access type

Each mapped range can be qualified as read-write, read-only, write-only, or restored to target memory.

Address translation

Overlay can be mapped using physical or logical code or logical data addresses. The mapper performs a logical-to-physical translation and uses physical addresses for all subsequent operation. Listings of overlay mappings are presented as physical ranges.

User Programmable Machine (UPM) A and B

Because of the complexity of address multiplexing, the SuperTAP is not able to map overlay memory to target memory that uses the UPM. The UPM is typically used to access target DRAM. If you want to map overlay memory to those regions where target DRAM is not functioning, you can reconfigure the UPM regions to use the general purpose chip-select machine (GPCM) instead. The GPCM is normally designed to be used with EPROM, ROM, and SRAM type devices. But you can switch the UPM regions to GPCM by changing the machine select (MS) bits in the chip-select BR*x* registers.

Parity

Overlay memory does not support parity.

External bus master

Overlay supports accesses by both synchronous and asynchronous external bus masters. When you want an external bus master to access overlay you must set two overlay options that control whether reads or writes go to target as well as overlay. The commands are **ovwritethru on** and

ovreadthru on. To prevent contention, you must also make sure that any target ROM device that is overlayed is deactivated.

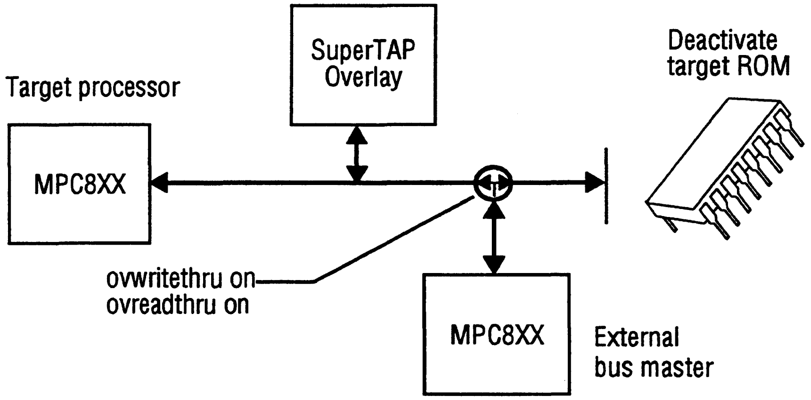


Figure 3-1 Overlay memory configured for external bus master

Saving and restoring overlay maps

Search

Keywords:
Maplist
Include

To enable recreation of debugging conditions, the **maplist** command saves current mappings as a command file that can be restored using **include**.

Note that overlay memory retains its content even if target power is lost.

How memory mapping is handled

This section summarizes the rules governing overlay function and use. To make effective use of overlay you should understand these basic principles.

If you want a more extensive explanation, these principles are fully illustrated in the three extended examples later in this chapter:

- Mapping physical memory
- Mapping logical memory

Mapping overlay is the process of requesting an overlay location for memory address ranges and their attributes. You use the **map** command to make these assignments, and **mapclr** and **maplist** to clear or display the current mappings.

Page allocation is the process the overlay system goes through to assign overlay mapping requests to one of the eight pages and maintain the integrity of memory accesses. Page allocation is determined by the physical address range of the map request.

Page addressing

Page starting and ending addresses are automatically aligned to boundaries corresponding to the page size. For example, the following command:

```
map 0x30000..0x31fff
```

will cause the overlay mapper to display a warning message, and then allocate a page of overlay to the 128K range 0x20000..0x3ffff.

```
maplist
MAP 0x00020000..0x0003FFFF=RW
```

Each mapping that falls outside a previously established range causes allocation of new overlay pages until the overlay resources are exhausted.

This processes is illustrated in detail in "Adding ranges to overlay" on page 3-10.

Characterizing additional attributes

If logical addressing is used, the logical range is converted to a physical range using the translation tables present in target or overlay memory. Full procedures and an illustration of mapping to logical addresses are provided in "Mapping overlay using logical addresses" on page 3-12.

Each individual range can be qualified to respond as read-write (**rw**), read-only (**ro**), or returned to target (**target**). Different access types can be located in the same bank of overlay.

Note

Overlay is designed to block target writes to read-only overlay during run. It does not halt emulation. To break on such illegal writes, set an access breakpoint over the range in question.

Basic procedures for mapping overlay

This section summarizes the procedures for typical overlay use. Extended examples follow and highlight special considerations and operational characteristics. Command descriptions and many of the procedures are also provided in Help.

Qualifying mappings for addressing mode

Search Keywords:
Space
Address

The debugger defaults to physical addressing mode. Use the **mode** option of the **map** command to override the defaults for a single mapping.

This procedure is covered in detail in the extended examples of mapping with logical addresses.

Adding an overlay mapping

Overlay mappings are entered from the command line using the **map** command line to enter either logical or physical address ranges:

Search Keywords:
MAP
Addressing mode
Overlay access types

- Enter the address range and attributes of the mapping:

```
map start..end | start..+length [,mode]
[=type]
```

For full procedures and command line options, see Help.

Modifying an existing overlay mapping

Overlay mappings are modified by re-mapping the existing range with its new attributes. In all cases, the new mapping replaces the previous one. If they overlap, the portion of overlap takes on the new attributes, but the overlay contents are maintained.

Restoring memory to target

Convert overlay mappings to target memory using the **map** command with the **=target** option.

➤ **To restore memory to target**

1. Use **maplist** to select the mapping to restore to target.
2. On the command line, enter the listed range or base plus length, and assign it to target:

```
map {start..+length | start..end}=target
```

3. Click Enter Command, or press <Return>.

Search Keywords:
MAP
MAPCLR
Target memory

If you use logical addressing, you can also enter the original logical range. The mapper will translate it to physical and restore control of the appropriate physical range to target.

Displaying/saving/restoring memory map

Use the **maplist** command to display the current mapped memory or to save that listing for future use.

➤ To display the current memory mappings

- From the Command window, type **maplist**

Search Keywords:
MAPLIST
Include

The current mappings display in the Command window. All mappings originally entered as logical ranges are displayed as their physical equivalents.

➤ To save current mappings to a file

- Type **maplist** *filename*

If you plan to use the file as a command file, give it an **.inc** extension. The current listing of physical ranges is saved to the named file in the current working directory. Provide a full path if you want to save it elsewhere.

➤ To restore a previously saved map file

- Use the **include** command to reload *filename*.inc

—or—

- At MWX-ICE startup, use the include option in the MWX-ICE Startup Options Editor to load *filename*.**inc.**

This restores the physical memory map.

Clearing the overlay memory map

Search Keywords:
MAPCLR

Use the **mapclr** command to clear all memory mappings and return memory control to the target.

Mapping physical memory

Many target designs employ physical addressing, rather than logical. This section illustrates how overlay is enabled for such a system and how memory resources are allocated.

Scenario

An example region of target memory has a physical base address of 0x8000 and a length of 32K (0x8000). Of the several modules located in this region, the module you are interested in starts at 0x90c4 and ends at 0x9fdf.

Procedures

- Map the range:

```
map 0x90c4..0x9fdf
```

The following occurs when you enter this mapping in a 1 MB overlay system:

- An entire 128K bank of overlay is allocated, starting at 0x0.
- Because neither the start nor the endpoint falls on a 128K boundary, the overlay mapper adjusts the range to 0x00000000..0x0001FFFF and warns that it has done so.

```
Problem found while configuring overlay memory.
The Overlay Map request starting and/or ending ad-
dresses were adjusted to a 128K byte boundary. Over-
lay can only be mapped to 128K byte regions. Display
current overlay mapping to see the adjustments made.
```

- The 128K range is enabled to respond to read-write accesses (the default).

Adding ranges to overlay

Although one 128K bank of overlay is allocated, you still have 896K of overlay memory available. You can add additional address ranges to overlay. Both read-only, write-only, and read-write mappings can co-exist within overlay memory.

For example, if you wanted to map over target ROM, located at 0x100000..0x00103CD0, you could add the following command:

```
map 0x100000..+0x1ffff=ro
```

The new mapping would be adjusted for boundaries, and enabled adjacent to the initial one in the same bank. A **maplist** display of mappings would show:

```
maplist
 MAP 0x00000000..0x0001FFFF=RW
 MAP 0x00100000..0x0011FFFF=RO
```

Overlapping requests

The mapper constantly re-evaluates mapping requests, consolidates overlapping or adjacent mappings. and adjusts boundaries. When a new mapping overlaps an existing range, the portion within the overlap takes on the attributes of the new mappings.

For example, a routine might be located from 0x10000 to 0x2ffff in read-only memory. If you add this new mapping (**map 0x10000..0x2ffff=ro**) to the ones above, the **maplist** shows the following:

```
maplist
 MAP 0x00000000..0x0003FFFF=RO
 MAP 0x00100000..0x0017FFFF=RO
```

Because the new overlay segment overlaps an existing one, and also cuts across the 128K minimum map size, two 128K segments are mapped as read-only (0x0..0x0003FFFF).

Now, if you map the lowest 128K as read-write memory, notice how the attributes change again.

```
map 0x0..0x1ffff=rw
```

```
maplist
 MAP 0x00000000..0x0001FFFF=RW
 MAP 0x00020000..0x0003FFFF=RO
 MAP 0x00100000..0x0017FFFF=RO
```

Mapping overlay using logical addresses

This section explains and illustrates how overlay is allocated and enabled for a target system that use the memory management unit and logical addressing of the processor.

Note

For a system-level discussion of logical addressing and MMU support, see “Support for the MMU and logical addressing” on page 2-28. Relevant portions of that chapter are expanded here for their application to overlay use.

Required target configuration

Before requesting overlay mapping using logical addressing, the target must meet certain requirements:

- Valid translation tables must be in memory before the instruction or data MMU is enabled.
- The MMU must be initialized.

Valid translation tables

Tables must be in memory before mapping can begin.

Initialized MMU

During mapping, the memory management unit must be initialized and translation tables must be loaded into memory before the mapper can perform logical-to-physical translation and enable regions of physical memory. The MSR_{IR} or MSR_{DR} bits must be configured to enable MMU translation.

Memory map

The mapper uses the following method during translation:

- The MMU is queried for the physical location of the logical starting address.
- The length of the mapping is inferred from the logical range entered.
- The request is adjusted to meet overlay granularity and alignment requirements.
- A continuous block of overlay is enabled beginning at the physical starting address and extending for the length of the range.

Note

If you don't know whether logical ranges translate to contiguous physical ones, or if logical to physical translations are readily available, use physical ranges when mapping overlay.

Understanding and using the maplist displays

The **maplist** displays the physical ranges created when the mapper translates the logical input. Consequently, you should understand the logical-to-physical mappings for the memory you are overlaying.

To keep track of how the mapper translates and adjusts the logical ranges to fit overlay granularity requirements, you can activate the **log** utility and capture the **maplist** reports as you make each entry:

1. Turn on the **log** utility, configured to log output to a file:

   ```
   log /a on="ovlmap"
   ```

   The /a filter appends successive output to the file.
2. Enter a **map** request.
3. Enter **maplist**.

4. Repeat 2 and 3 until all memory is mapped.
5. Open a new shell and display the file, or print it.

Like **maplist** files, **log** files can be edited and used as command files to re-map overlay.

Warnings and error messages

The overlay mapper automatically adjusts the start/end points of each map request to force them onto 256-byte boundaries and to adjust the bank's start/end points to make most efficient use of overlay resources.

The mapper prompts you when a mapping cannot be completed as requested and warns you when it makes changes to the map request you have entered:

"Incorrect syntax. . ./Symbol not found": You have mis-keyed the entry or used illegal syntax.

"Overlay request endpoints adjusted": Each overlay mapping is adjusted to begin and end on a 128K boundary (0x0 and 0x1ffff) and to span an appropriate multiple of 128K ranges.

Loading code and copying memory

The **map** command simply defines where memory accesses take place (target is the default) and assigns access type attributes to the memory region. You use **load** or **dnl** to load code into target or overlay memory, and **upl** to save the contents of target or overlay to a named file on the host. The **overlay** and **copy** commands enable you to copy the contents of memory between target and overlay.

Downloading code in ELF/DWARF format

Search Keywords:
LOAD
File format

If your object code is in ELF/DWARF format, choose Load from the File menu, or use the **load** command with the Command window to download to target and overlay:

```
load demo\cdemon.elf
```

Depending on any overlay mapped, this routes code to appropriate overlay and target memory locations. Filters are provided to select whether to load symbols, set the program counter, append to exiting code, etc.

Downloading other formats

Search Keywords:
DNL
File format
DNLFMT

If code is not in ELF/DWARF format, use the **dnl** command to download. You can use the **dnl** command to download a hex file from the host to the target in the format specified by the **dnlfmt** command. Before downloading, memory must be qualified using the code field of the **address** and **space** commands.

For example, to download an S-record file named *main.srec* using physical addressing:

```
address code physical
dnlfmt srec
dnl "main.srec"
```

Note

When **dnl** and **upl** are used, the file being transferred acquires the object format specified by **dnlfmt** or **uplfmt** (S-Record is the default). Debugging with symbols is supported only for ELF/DWARF format.

Stopping a download

If for any reason you decide to stop a download while it is in progress, click the Stop button.

Copying memory contents between target and overlay

The contents of overlay and target memory may be copied in either direction to identical or different memory locations. The following procedures are most useful when you need to copy the contents of your target ROM or PROM into overlay memory for patching, to avoid having to burn a new ROM.

Specifying source and destination

Search Keywords:
OVERLAY
Command
Copying memory
COPYTO,

Two settings in the **overlay** matrix, **copyfrom** and **copyto**, specify source and destination, respectively. The **on** option chooses overlay; **off** selects target.

➤ **To prepare to copy from target to overlay**

1. Select target as the **copyfrom** source:

```
overlay copyfrom off
```

2. Select overlay as the **copyto** destination:

```
overlay copyto on
```

Performing the Copy operation

If you want point-and-click access to the MWX-ICE copy functions, use the Copy page of the Memory Commands notebook. Otherwise use the **copy** command from the command line of the command window.

`map 0x0..0x1ffff=ro`	Enables overlay memory to respond to the specified range as read-only memory.
`overlay copyfrom off`	Specifies target as source of copy.
`overlay copyto on`	Specifies overlay as destination of copy.
`copy 0x0..0x1fff,0x0`	Specifies target range to copy to overlay and overlay starting address.

Saving overlay to a file

You can save a portion or all overlay or target memory to a named hex file using the **upl** command. The format of the files depends on the setting of the **uplfmt** command. Symbols are not supported.

Search

Keywords:
UPL
UPLFMT
Memory access qualification

Before uploading the file, set the memory characteristics using the **address** and **overlay** commands. Overlay memory does not qualify accesses for size.

For example, suppose the contents of overlay memory from 0x..0xffff have the address mode, physical. Enter the following:

```
overlay read on
address read physical
upl "myfile.hex", 0x0..0xffff
```

Controlling the source of accesses

Normally when overlay is mapped, you want all accesses to the mapped region to occur in overlay. To refine and occasionally override the memory mappings, use the **overlay** command. For the 14 types of access listed, you can toggle the source of access individually from overlay to target.

`ASM`	`Destination of line assembly using ASM`
`AUX`	`Location of SIT mode`
`CODE`	`Code window display and accesses using LOAD and DNL`
`COMP1`	`Source of first argument of COMPARE`
`COMP2`	`Source of second argument of COMPARE`
`COPYFROM`	`Source memory for a COPY`
`COPYTO`	`Destination memory for a COPY`
`DIAG`	`Memory for use with DIAG`
`FILL`	`Destination of FILL`
`READ`	`Generic reads using DUMP, CRC,UPL, DISASSEMBLE`
`SEARCH`	`Memory for use with SEARCH`
`STACK`	`Memory accesses for the stack display`
`TEST`	`Memory for use with TEST`
`WRITE`	`Memory for generic writes (SETMEM)`

The default for each type is overlay (**on**), unless changed from the command line or in the Memory dialog of the Emulator Configuration window. If the debugger find no overlay mapped for the type of access, it reverts to target. However, if a type is set to target (**off**), the debugger does not access overlay, even if it is mapped.

For example, if you want all diagnostics to run in target memory, set **overlay** as follows:

```
overlay diag off
```

Additional information

- The tutorial in Chapter 10 offers additional practical examples.
- Help provides detailed descriptions of all the commands mentioned in this chapter, as well as structured "browse sequences" that organize the Help topics by subject.

Chapter 4

Programming Flash Memory

This chapter describes how you can use MWX-ICE and the SuperTAP to program, lock, and erase the flash memory in your target system.

Overview

Flash memory is widely used in embedded system designs because of its non-volatility, high-performance, low-cost. MWX-ICE and the SuperTAP provide a fast and efficient means of programing and erasing flash memory devices in your target system. When you use the SuperTAP, there is no need to use separate flash programing tools, and no need to connect additional hardware to your target system. You can prototype and debug your system design all within the MWX-ICE environment.

Using special flash memory commands, you can configure, erase, program, and lock and unlock flash memory devices.

To avoid inadvertently writing to flash memory, you can only program, erase, or lock one flash memory device at a time. If you are programming more than one device, you need to *remove* the existing device, before you can configure the next. Removing the device only disables the capability of programming the device. The actual device and its memory contents are not changed in any way.

Example of a target system using flash

The example shown in Figure 4-1 shows a target system that uses three flash memory components. These are 512K by 8-bit devices. Two of the flash memory components are linked in parallel, and are accessed by chip select 0 (CS0), which uses a 16-bit port. The base address is 0x0. The third flash memory chip is located at address 0x80000000. This address is accessed by chip select 1 (CS1), using an 8-bit port.

Programming the flash for this target system requires two separate steps. First, you configure and program the devices at address 0x0. The two AMD29F040 memory chips are configured and programed together. The command to configure the device is **amd29f040(0x0, 16)**. This tells the debugger the type of device (29f040), the base address (0x0), and the width for memory access (16). Next you erase the device and then program it by downloading code.

Once you've programmed the flash at address 0x0, you need to remove that device from the configuration table using the **RemoveDevice()** command. You need to remove the existing device, so you can configure the flash memory that starts at address 0x80000000. The command to configure the remaining flash chip is **amd29f040(0x80000000, 8)**. This tells the debugger the address of the next device, and that the access is now 8-bit.

Using MWX-ICE and the SuperTAP, you can expect to program a 512K flash component in less than five minutes.

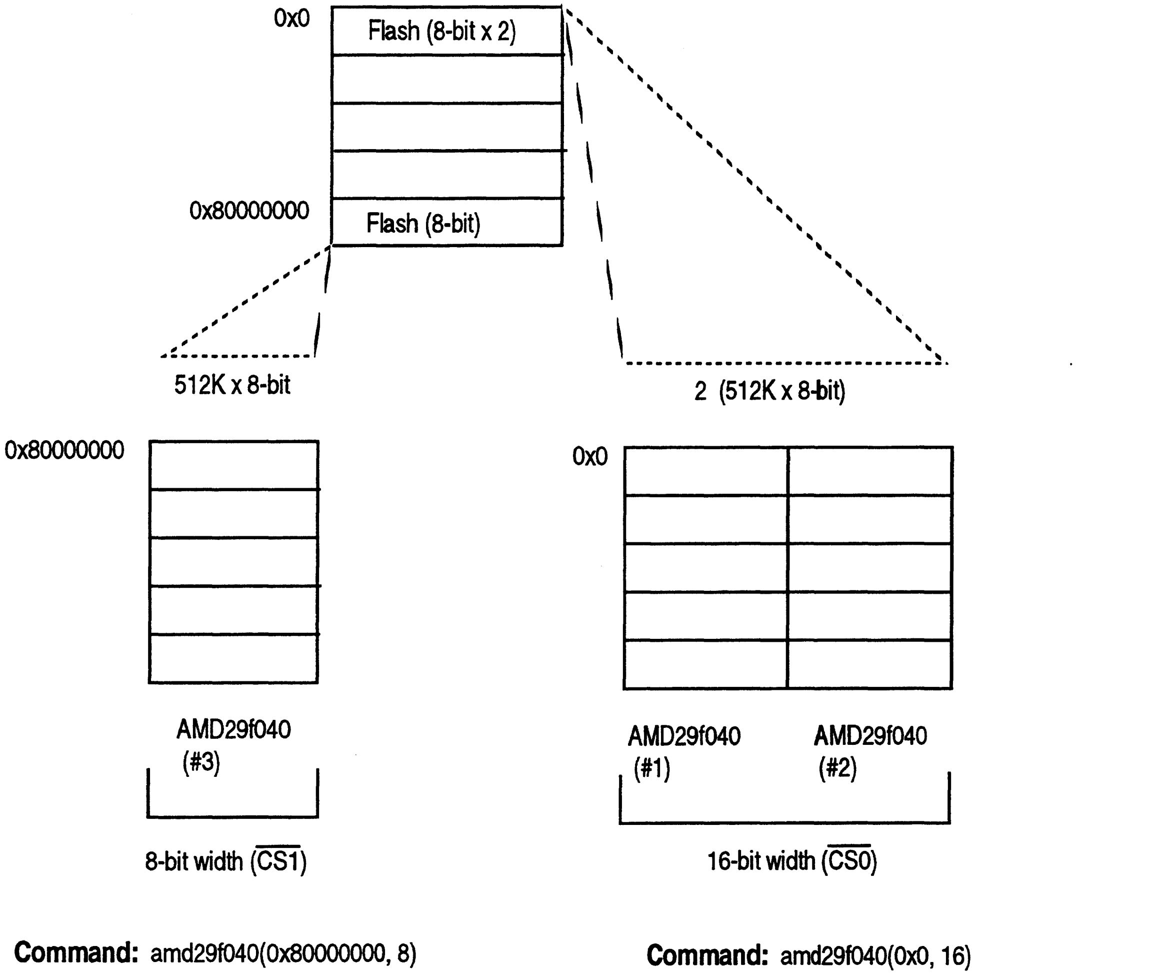

0x0
Flash (8-bit x 2)
0x80000000
Flash (8-bit)
512K x 8-bit
2 (512K x 8-bit)
0x80000000
0x0
AMD29f040
(#3)
AMD29f040
(#1)
AMD29f040
(#2)
8-bit width ($\overline{CS1}$)
16-bit width ($\overline{CS0}$)
Command: amd29f040(0x80000000, 8)
Command: amd29f040(0x0, 16)

Figure 4-1 Example showing commands to configure flash memory devices

How to program your flash memory

Programming flash memory is easy using MWX-ICE macros and commands. Once you run the macros, you can program flash using the MWX-ICE **load** and **dnl** commands.

Basic procedure

Using MWX-ICE and the SuperTAP to program the flash memory components in your target is relatively straight forward. The basic procedure includes the following steps:

- ❑ Including the flash programming macros.
- ❑ Configuring the device.
- ❑ Erasing the contents of the component.
- ❑ Programming the device by downloading code.
- ❑ Locking the device to prevent accidental writes.

These steps are described on the following pages.

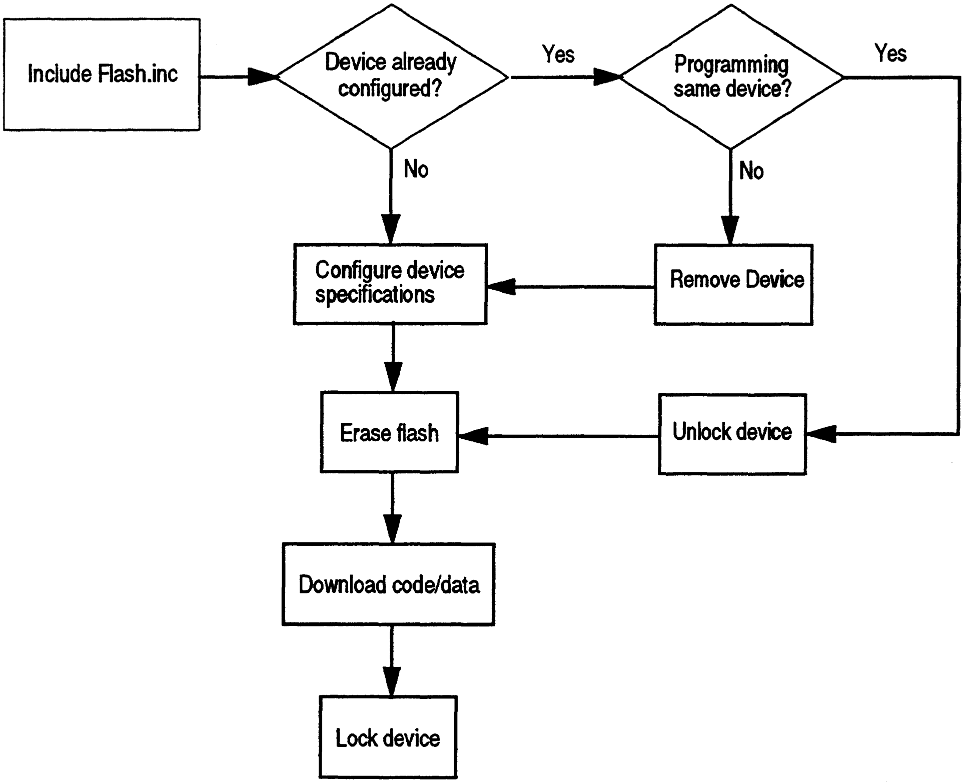


Figure 4-2 Flow chart showing MWX-ICE flash programming procedure

Including the flash programming macros

Before you can program, erase, or lock the flash memory in your target, you need to set up the programming macros. These macros are in a file called Flash.inc. To set up the macros, you must direct MWX-ICE to process the file containing the macros. You need to have this file included whenever you wish to program flash, so you may want to include as part of your startup configuration.

The macros are in a file called Flash.inc, which is located in the *installdir*\amc\st8xx directory.

➤ To setup the flash memory macros:

1. Start MWX-ICE.
2. From the File menu, choose Include Commands.
3. In the Windows file browser, click the amc directory, and then click the st8xx directory.
4. Select Flash.inc and click OK.

   The Command window displays the macros as they are read. It may take awhile for the file to be processed. When the include file is finished, the Command window stops scrolling.

   For information about including command files at startup, click Help from the Startup Options Editor.

Configuring flash memory components

The first time you program flash using MWX-ICE, you need to configure the device specifications. MWX-ICE needs to know the type of device, the base address, configuration, and virtual width of the flash memory device. Note that you can only configure and program one device at a time. If you have already configured a flash memory component, you need to unlock it so you can program it again, or you need to remove it, if you want to program a different device.

➤ To configure the flash device

1. Be sure to include the flash programming macros. See page 4-6.
2. Specify the base address, configuration, and virtual width of the flash memory device, using the AMD or Intel macro provided for your device.

   For the list of supported components and their configuration commands see page 4-12.

   For example, for an Am29F040 with a base address of 0x0 and a width of 16, enter:

   ```
   Amd29f040(0x0,16)
   ```

   For some components, such as the Intel 28F200, which can be configured as either an 128K x 16, or 256K x 8 device, you enter the base address, and the input/output architecture (8 or 16), as well as the width for memory access.

   ```
   Intel28f200(0x0, 16, 16)
   ```

3. Once you've configured the device, you need to erase the contents:

   ```
   EraseDevice()
   ```

Note

Once write protection to the flash device has been disabled (by using the Amd*device,* Intel*device*, or UnlockDevice() macros, all writes in the device address range are programmed. To avoid inadvertently writing to your flash devices, be sure to disable flash programming by using the LockDevice() macro once programming is complete.

Downloading code to flash memory

You program the flash by downloading code to target. It may take several minutes to download the code to flash. Once the device has been configured, all writes in the address range of the device go to the flash component. For this reason, you need to lock the device when you are finished programming.

➤ **To program the flash device**

1. Program memory using the **load** or **dnl** commands.

   For example, to load the Intel hex file, myfile, enter:

   ```
   dnlfmt intel
   dnl "myfile"
   ```

   Note: You may need to change to your working directory.

2. Once you've programmed the flash, you need to disable the flash-programming feature so that flash memory is not inadvertently programmed:

   ```
   LockDevice()
   ```

Programming flash that has already been configured

If you want to program flash that has already been configured, and you are programming the same device with the same base address and width, you don't have to configure it again. You just need to unlock it and clear the memory contents.

If you have already included the flash support file and are programming the same device with the same base address and width

➤ To open the device for programming

1. Enable flash programming:

   ```
   UnlockDevice()
   ```

2. Erase the device:

   ```
   EraseDevice()
   ```

3. Program memory using the **load** or **dnl** commands.

   For example, to load the Intel hex file, myfile, enter:

   ```
   dnlfmt intel
   dnl "myfile"
   ```

   Note: You may need to change to your working directory.

4. Once you've programmed the flash, you need to disable the flash-programming feature so that flash memory is not inadvertently programmed:

   ```
   LockDevice()
   ```

Programming another flash component

If you have already included the flash support file and are programming a different device or the same device type with a different base address or width, you first need to remove the existing configuration.

➤ To remove a device

- Use the following command.

```
RemoveDevice()
```

Once you've removed the existing device specification, you can configure another one, see

- ❑ "Configuring flash memory components" on page 4-8.
- ❑ "Downloading code to flash memory" on page 4-9.

Flash memory support

The following flash memory devices can be programmed using SuperTAP and MWX-ICE. The following tables show the name of the component, and the syntax of the command used to specify the flash memory configuration.

AMD flash memory components

Component	Configuration command
29f010	Amd29f010(*base_address*, *width*)
29f016	Amd29f016(*base_address*, *width*)
29f040	Amd29f040(*base_address*, *width*)
29f080	Amd29f080(*base_address*, *width*)
29f100	Amd29f100(*base_address*, *by8or16*, *width*)
29f200	Amd29f200(*base_address*, *by8or16*, *width*)
29f400	Amd29f400(*base_address*, *by8or16*, *width*)
29f800	Amd29f800(*base_address*, *by8or16*, *width*)

Intel flash memory components

Component	Configuration command
28f001	Intel28f001(*base_address*, *width*)
28f002	Intel28f002(*base_address*, *width*)
28f004	Intel28f004(*base_address*, *width*)
28f008SA	Intel28f008SA(*base_address*, *width*)
28f400	Intel28f400(*base_address*, *by8or16*, *width*)
28f200	Intel28f200(*base_address*, *by8or16*, *width*)
28f016SA	Intel28f016SA(*base_address*, *by8or16*, *width*)
28f032SA	Intel28f032SA(*base_address*, *by8or16*, *width*)

Configuration command syntax

Argument	Description
base_address	The address where the flash component is located. The address must be expressed in hexadecimal.
by8or16	Used for those components that provide user selectable 8- or 16-bit operation. **Use: For:** 8 x8 devices. 16 x16 devices.
width	The width of the memory access. For example, four x8 devices can be linked in parallel to create a region of 32-bit memory. This region could then be accessed by a chip-select using a 32-bit port. **Use: For:** 8 8-bit width. 16 16-bit width. 32 32-bit width.

Flash memory macros

The macros for programming flash memory are listed in the following table. The macros are also described in Help. Just like MWX-ICE commands, you invoke the macros in the Command window. Note, however, that macros behave a little differently from MWX-ICE commands:

- Macros are case-sensitive.
- You must type the opening and closing parentheses () even if the macro takes no arguments.
- The Flash programming macros are only available if you've included the Flash.inc file.

Macro	Description
Amd*device(base_addr, width)*	Specifies the base address and width of flash memory to be programmed.
Amd*device(base_addr, by8or16, width)*	
Intel*device(base_addr, width)*	Specifies the base address, configuration, and width of flash memory to be programmed.
Intel*device(base_addr, by8or16, width)*	
EraseDevice()	Erases the device specified by Amd*device* or Intel*device*.
LockDevice()	Disables flash programming.
RemoveDevice()	Removes existing flash device specification, allowing new devices to be specified.
UnlockDevice()	Enables flash programming.

Macro names

If you want to rename the macros, edit the Flash.inc file, after first making a backup copy.

Note: The emulator variables in the Flash.inc file are intended to be used only by the macros.

Chapter 5

Tracing Program Execution

Using the trace features of MWX-ICE, you can capture and record, in real time, the execution history of the processor as SuperTAP executes the target program. Using trace history, you can verify the correct performance of the software and hardware, and find errors that may occur in the program's execution.

Note

All of the commands mentioned in this chapter are fully explained in the command reference provided in Help.

What is trace?

The SuperTAP emulator uses a sophisticated system to record the bus activity and execution history of the target application in realtime. This information is stored in the emulator's trace buffer. Trace accumulates on every clock cycle and is pipelined to qualify cycles for retention and to synchronize timing with the emulator's event system.

The emulator's trace buffer can store approximately 32,000 *frames* of trace. A trace frame is like a frame from a motion picture film: the frame shows the state of the processor activity at each clock cycle. When you view the trace frames together, you get a complete history of program execution.

Each trace frame contains 128 bits of information. Using emulator trace capture variables and the event system, you can selectively filter the kinds of processor activity you wish to capture. Once trace has been captured, you can use trace display variables to view the contents of the trace buffer in several different ways. You can view trace in either the Emulator Trace window or in the Command window.

How you can use trace

The tutorial in Chapter 10 includes several expanded examples of trace capture and display. You may wish to work through one or all of them to familiarize yourself with the features at your disposal. The following section describes some of the more typical uses of the emulator's trace features.

Qualify trace using the event system

Trace can be captured so that only the activity of interest is retained. This *qualified* trace capture maximizes the available trace buffer and saves you time.

Execute and trace power-up or reset sequences

You can a execute a target power-up or reset sequence while collecting trace history. With this you can more easily debug startup code.

View trace while running

The SuperTAP emulator allows you to view and upload trace history without stopping or even pausing emulation. This means you can view your program's activity without disturbing its real-time operation.

Analyze timing

Timestamp information provides a quick, accurate way to measure time spent during specified portions of target execution.

Save trace history for analysis

You can easily store trace history in a file. You can specify a journal or log file in an MWX-ICE notebook, then simply display trace. You can even log trace to a file *without* stopping emulation. After trace is saved, you can edit the file to add comments for future reference. For example, comments may be added to aid in documenting failure conditions.

You can use either the **log** or **journal** commands or the guided configuration provided by the Debugger Files Notebook.

View trace while running

SuperTAP allows you to view and upload trace history without stopping or even pausing emulation. This means you can view your program's activity without disturbing its real-time operation.

Preparing to capture trace

When enabled, the trace system uses the captured raw bus or clock cycles as the basis for trace display. Several configuration variables are provided to specify when to enable and disable trace and which cycles to collect in the trace buffer.

The following sections provide standard switch settings for each of the most common types of trace capture. Settings are expressed in a manner suitable for use as command files. A command file is simply a file containing debugger commands that will be executed when the file is loaded using the debugger's **include** command. Such files can be very useful in eliminating frequently duplicated keystrokes and actions.

The settings shown are by no means the only possible combinations. In many cases, these basic settings can be combined; for example, those for trace disassembly and continuous tracing during run.

Each switch is a separate command. See the Help for a complete description of each command syntax, options, and limitations.

Starting with an empty trace buffer

The contents of the trace buffer are retained between sessions unless the emulator is powered down or reset. In a multi-user environment or when starting a new debugging session, always clear the trace buffer before beginning to capture new trace. In addition, a change to **trqual** (see below) alters the trace capture criteria. Anytime you change **trqual**, you should clear trace before proceeding.

➤ To clear the trace

- Switch to the Command window. In the Enter Command box, type **trclr** and click Enter Command.

  —or—

  Switch to the Emulator Trace window. From the Actions menu, choose Clear Trace.

Setting the trace capture variables

Search Keywords:
TRSYS
TRACE
TRQUAL
SIGA_MUX
SIGB_MUX
TIMCLK
TRINIT
TRRUNCLR
TRCEXT
PPT

Ten trace variables govern whether and how trace is captured. They must be set before you begin.You can set these variables in the Trace configuration dialog box or from the Command window.

To open the Trace dialog box

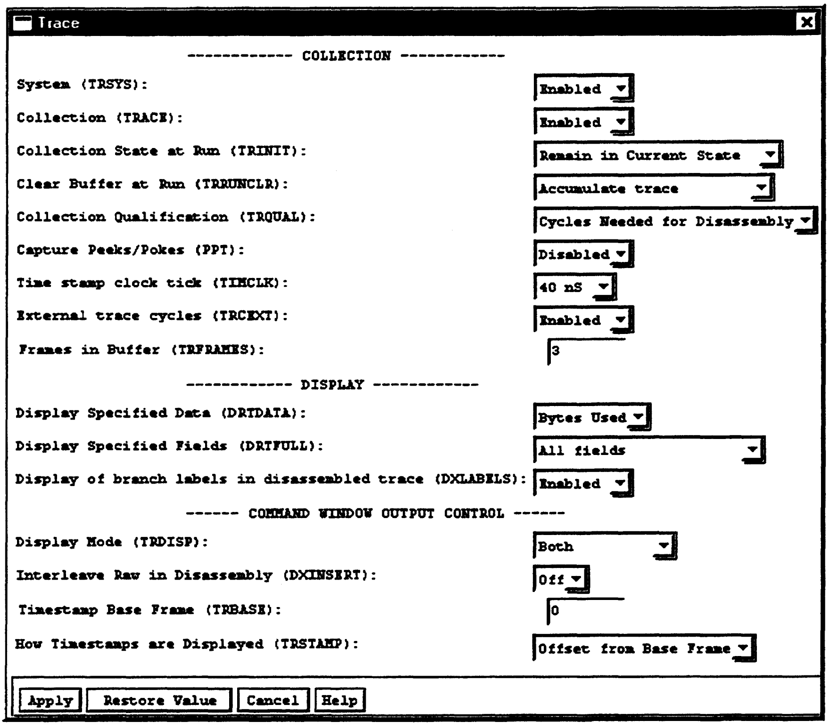
1. From the Displays window, choose Emulator Configuration.
2. In the Emulator Configuration window, click Trace.

Because the SuperTAP multiplexes some signals to the event and trace systems, you need to select which of these signals to record in trace. You can open the Event configuration dialog box, or use the **siga_mux** and **sigb_mux** commands to select the signals.

Trace Capture Options
Set these before running code.

Trace Display Options
These options control the display of trace in the Emulator Trace Window and in the Command window.

Trace Display Options
These options control the display of trace in the Command window only.



------------ COLLECTION ------------

System (TRSYS):	Enabled
Collection (TRACE):	Enabled
Collection State at Run (TRINIT):	Remain in Current State
Clear Buffer at Run (TRRUNCLR):	Accumulate trace
Collection Qualification (TRQUAL):	Cycles Needed for Disassembly
Capture Peeks/Pokes (PPT):	Disabled
Time stamp clock tick (TIMCLK):	40 nS
External trace cycles (TRCEXT):	Enabled
Frames in Buffer (TRFRAMES):	3

------------ DISPLAY ------------

Display Specified Data (DRTDATA):	Bytes Used
Display Specified Fields (DRTFULL):	All fields
Display of branch labels in disassembled trace (DXLABELS):	Enabled

------ COMMAND WINDOW OUTPUT CONTROL ------

Display Mode (TRDISP):	Both
Interleave Raw in Disassembly (DXINSERT):	Off
Timestamp Base Frame (TRBASE):	0
How Timestamps are Displayed (TRSTAMP):	Offset from Base Frame

Figure 5-1 The Trace dialog box.

➤ To set trace options

- In the Trace dialog box, select the settings for each option, then click the Apply button.

  You can leave the Trace dialog box open while you create your event system statements.

  —or—

- From the Command window, type the name of the command and press return.

  The following tables lists the trace capture commands.

Command	Description
trsys	Enables the trace subsystem. It should be **on** at all times.
trace	Enables trace capture. It should be **on** unless you are using the event system's trace control actions to qualify trace.
trinit	Selects the initial state of trace capture at run. It should be **on** unless you are using the event system's trace control actions to qualify trace.
trrunclr	Clears (**on**) or appends (**off**) trace to the current trace buffer at each run.
trqual	Determines whether bus or clock cycles—or only enough cycles for disassembly—are captured. For disassembly, **trqual** must be set to **dxqual**. You should always clear the trace buffer after changing the **trqual** value.
siga_mux	Selects which multiplexed signals to record in trace (irq[0, 1, 7] or lsa[0, 1, 7]).
sigb_mux	Selects which multiplexed signals to record in trace (irq[2:6], lsa[2:6], wp, pcmcia, dp, rsv).
trcext	Enables capture of external bus cycles.
ppt	Enables capture of peeks and pokes during pause. Typically left **off**. Peek/poke cycles are not shown in disassembled trace.
timclk	Selects timestamp clock resolution (40ns, 200ns, 1us, 10us, 100us, 1ms, 10ms, or 100ms).

See "Capturing trace suitable for disassembly" on page 5-10 for additional variables that must be set for trace disassembly. Each of these variables is described in detail in Help.

Capturing continuous raw trace during run

Search Keywords: Raw trace

The most common configuration for trace capture is to capture enough cycles for trace disassembly at all times during run. This way, you can look at raw trace as well as disasssembled trace. As the trace buffer becomes full, newly captured trace frames overwrite the oldest frames. At each pause to run transition, new trace is appended to the trace buffer.

Set	To do this...
`trsys on`	Turn trace system on.
`trace on`	Turn trace capture on.
`trinit on`	Select initial trace state at run.
`trqual dxqual`	Capture enough cycles for disassembly.
`trrunclr off`	Append to buffer at return to run.

You may want to toggle **trqual** to **clock** to capture a frame of trace for each processor clock cycle.

Clearing raw trace at each return to run

Search Keywords: Trace clearing

If you want to capture raw trace and start fresh at each return to run mode, use the **trrunclr** switch. You can also clear trace manually during **drun** or pause mode using **trclr**.

Set	To do this...
`trsys on`	Turn trace system on.
`trace on`	Turn trace capture on.
`trinit on`	Select initial trace state at run.
`trqual dxqual`	Capture enough cycles for disassembly.
`trrunclr on`	Clear buffer at return to run.

Stopping trace when the trace buffer is full

Search Keywords: Trace control

If you want to prevent overwriting trace, configure the switches to turn off trace when the buffer is full. This setup requires that you save or clear trace at each pause, and restore the trace switch to its **on** setting.

Set	To do this...
`trsys on`	Turn trace system on.
`trace on`	Turn trace capture on.
`trinit current`	Use previous state at return to run.
`trqual dxqual`	Capture enough cycles for disassembly.
`trrunclr off`	Clear buffer at return to run.
`when trfull then troff`	Turn off trace when buffer is full.

Options include adding a break action to **troff** so that emulation stops when the buffer is full. If you know that you will preserve any needed trace at each pause, you might change **trinit** and **trrunclr** to **on**. This empties the trace buffer and turns on trace each time the emulator returns to run.

Capturing peek/poke activity

Search Keywords: Tracing peeks/pokes

Peeks and pokes are reads and writes that you or the emulator performs in memory during pause or emulation.

Set	To do this...
`trsys on`	Turn trace system on.
`trace on`	Turn trace capture on.
`trinit on`	Select initial trace state at run.
`trqual bus`	Capture **bus** cycles.When you use **dxqual**, only non-peek/poke trace will is disassembled.
`trrunclr off`	Clear buffer at return to run.
`ppt on`	Include peek/poke activity.

Qualifying trace using the event system

Search Keywords: Trace qualification

Certainly the most efficient use of trace is to qualify it using the event system. This ensures that only the activity associated with specified events is preserved.

Set	To do this...
`trsys on`	Turn trace system on.
`trace off`	Turn trace capture off.
`trinit current`	Select initial trace state at run.
`trqual bus`	Capture **bus** cycles. Use **dxqual** if trace will be disassembled.
`trrunclr off`	Clear buffer at return to run.
when add==*addr* then tron	Turn on trace with selected event.
when add==*addr* then troff	Turn off trace with selected event.
	—or—
when *event* then trone	Capture single cycle of interest.

If you plan to disassemble trace qualified by the event system, use the settings shown below for disassembly. Combine the trace control event statements with the trace system control settings.

Capturing trace suitable for disassembly

Search Keywords: Disassembly setup

To disassemble the raw trace cycles, the trace system requires a certain continuity of execution flow. Without it, the disassembler cannot reconstruct program execution.

To be able to disassemble trace, you must ensure that the processor show cycles are enabled so that disassembler can follow indirect branch instructions In addition, **trqual** must be set to **dxqual** to ensure that the cycles needed for disassembly are captured.

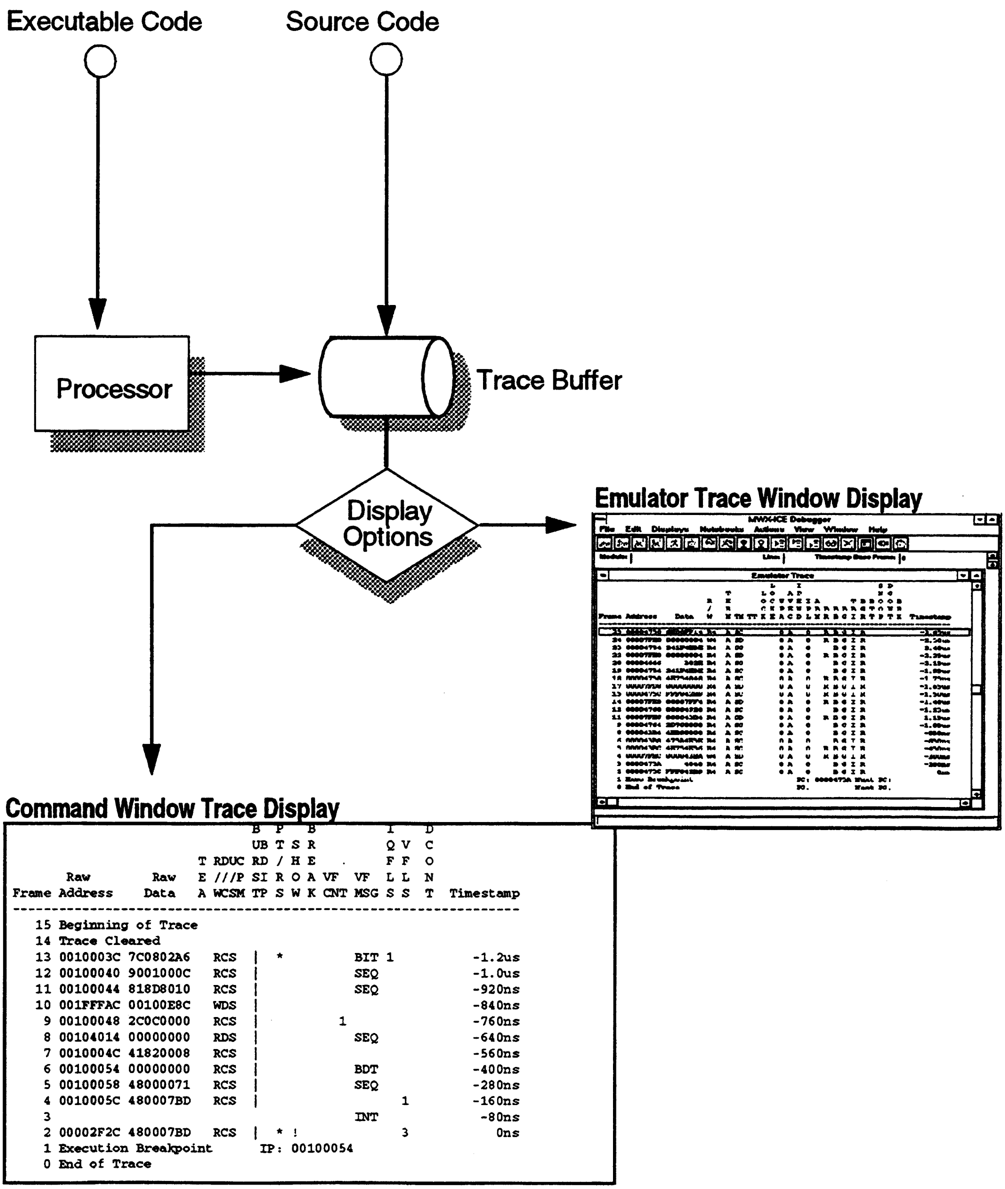
Set	To do this...
`showinst indirect`	Enable show cycles for indirect change of flow.
`trsys on`	Turn trace system on.
`trace on`	Turn trace capture on.
`trinit current`	Select initial trace state at run.
`trqual dxqual`	Capture enough cycles to allow disassembly.
`trrunclr off`	Clear buffer at return to run.
`dxlabels on`	Show symbols for branch destinations (only effective if MMU performs 1:1 logical-to-physical translation, or if the MMU is disabled).

Choosing the trace display interface

You can use one or both of the system's trace interfaces:

- □ The Emulator Trace window is designed for interactive, point-and-click use of the information in the trace buffer.
- □ Trace can also be displayed in the Command window. Command window display enables you to combine trace display with command-line controls, macro routines, and the system's **log** and **journal** utilities.

Both interfaces are described in the following sections.


Executable Code
Source Code
Processor
Trace Buffer
Display Options
Emulator Trace Window Display
MWX-ICE Debugger
File Edit Displays Notebooks Actions View Window Help
Emulator Trace
Command Window Trace Display

```
                               B P   B         I    D
                              UB T S R         Q V  C
                       T RDUC RD / H E   .     F F  O
       Raw        Raw  E ///P SI R O A VF  VF  L L  N
Frame Address    Data  A WCSM TP S W K CNT MSG S S  T  Timestamp
----------------------------------------------------------------
   15 Beginning of Trace
   14 Trace Cleared
   13 0010003C 7C0802A6  RCS  |  *         BIT 1          -1.2us
   12 00100040 9001000C  RCS  |            SEQ            -1.0us
   11 00100044 818D8010  RCS  |            SEQ            -920ns
   10 001FFFAC 00100E8C  WDS  |                           -840ns
    9 00100048 2C0C0000  RCS  |          1                -760ns
    8 00104014 00000000  RDS  |            SEQ            -640ns
    7 0010004C 41820008  RCS  |                           -560ns
    6 00100054 00000000  RCS  |            BDT            -400ns
    5 00100058 48000071  RCS  |            SEQ            -280ns
    4 0010005C 480007BD  RCS  |                  1        -160ns
    3                                      INT             -80ns
    2 00002F2C 480007BD  RCS  |  * !             3           0ns
    1 Execution Breakpoint     IP: 00100054
    0 End of Trace
```

Figure 5-2 The emulator's dual-interface trace system

Using the Emulator Trace window

Use the Emulator Trace window as your primary interface to the trace system. With simple mouse clicks, you can

- Scroll through the trace buffer.
- Change the trace display configuration.
- Change the timestamp format and offset base frame.
- Set breakpoints using trace information.
- Set the current scope based on the trace frame.
- Search for any string pattern in trace.
- Clear trace.

➤ To open the Emulator Trace window

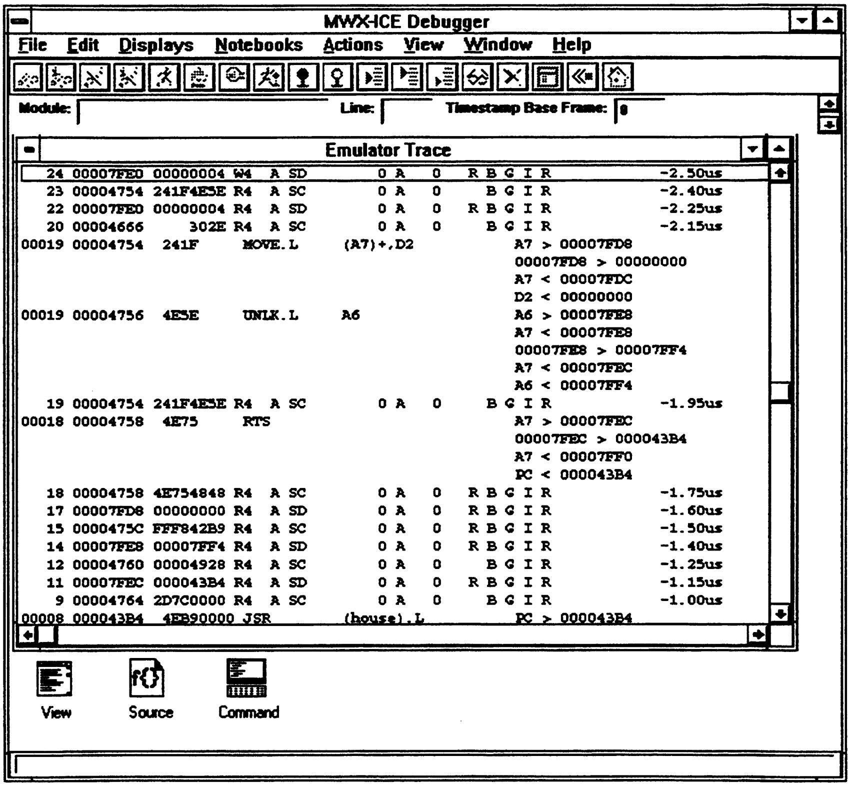
- From the Displays menu, choose Emulator Trace.

  The Emulator Trace window opens and displays the most recent screen of trace.

The Emulator Trace window displays a continuous buffer that is updated each time the emulator enters pause or when you perform actions that affect trace. To navigate trace, you can scroll the buffer or use the controls described in the next sections.

Note

When the Emulator Trace window's display option is set only to Raw, the window may take some time to refresh. The trace system is scanning backwards through the entire trace buffer and discarding invalid cycles. You can interrupt this processing by clicking the Stop button. If the trace buffer contents are complex, use the Command window for a faster trace display.



Module: Line: Timestamp Base Frame: 8

```
   24 00007FE0 00000004 W4  A SD        0 A   0   R B G I R           -2.50us
   23 00004754 241F4E5E R4  A SC        0 A   0     B G I R           -2.40us
   22 00007FE0 00000004 R4  A SD        0 A   0   R B G I R           -2.25us
   20 00004666     302E R4  A SC        0 A   0     B G I R           -2.15us
00019 00004754  241F     MOVE.L     (A7)+,D2            A7 > 00007FD8
                                                        00007FD8 > 00000000
                                                        A7 < 00007FDC
                                                        D2 < 00000000
00019 00004756  4E5E     UNLK.L     A6                  A6 > 00007FE8
                                                        A7 < 00007FE8
                                                        00007FE8 > 00007FF4
                                                        A7 < 00007FEC
                                                        A6 < 00007FF4
   19 00004754 241F4E5E R4  A SC        0 A   0     B G I R           -1.95us
00018 00004758  4E75     RTS                            A7 > 00007FEC
                                                        00007FEC > 000043B4
                                                        A7 < 00007FF0
                                                        PC < 000043B4
   18 00004758 4E754848 R4  A SC        0 A   0   R B G I R           -1.75us
   17 00007FD8 00000000 R4  A SD        0 A   0   R B G I R           -1.60us
   15 0000475C FFF842B9 R4  A SC        0 A   0   R B G I R           -1.50us
   14 00007FE8 00007FF4 R4  A SD        0 A   0   R B G I R           -1.40us
   12 00004760 00004928 R4  A SC        0 A   0     B G I R           -1.25us
   11 00007FEC 000043B4 R4  A SD        0 A   0   R B G I R           -1.15us
    9 00004764 2D7C0000 R4  A SC        0 A   0     B G I R           -1.00us
00008 000043B4  4EB90000 JSR        (house).L           PC > 000043B4
```

Figure 5-3 Emulator Trace window showing mixed with raw and assembly trace

Configuring the trace display

You can configure the trace display using the View and Actions menus and on the settings of several trace display variables.

View menu commands

The View menu provides options to control the primary display features. You can select the current display format to be any combination of raw trace, assembly, or source code.

Show raw trace This option displays bus or clock cycles and provides a header that identifies the address, data, and control signals. If assembly or source trace is also selected, the header information does not appear.

Show assembly trace Shows assembly language instructions. The raw trace frames are converted to their corresponding assembly-level instructions.

Show source trace Shows C or C++ source code. If it is available, the source code matching the raw trace frames is shown.

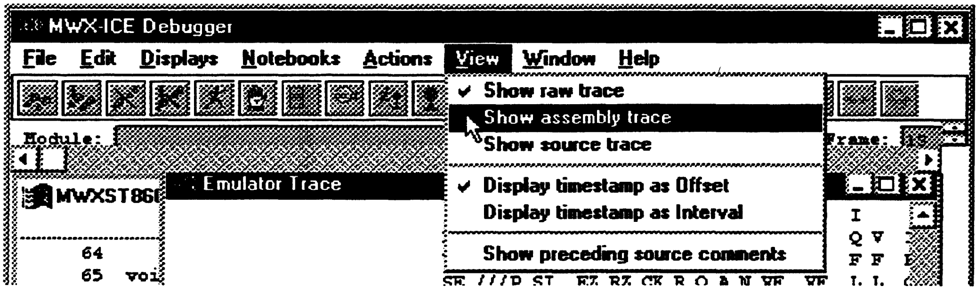


Figure 5-4 View menu display options

Note

When the Emulator Trace window is active (open or minimized), its display settings must be compatible with the setting of **trqual**. Insertion of assembly and source information is a feature of the trace disassembler and requires that trace has been captured with **trqual** set to **dxqual** or **clock**. Incompatible settings result in an error message.

Search Keywords:
Raw trace
Disassembled trace

For detailed information about the components of raw, disassembled, or source trace, see the tables beginning on page 5-33.

Display timestamp as Offset If offset mode is enabled, timing is offset from the base frame selected with the Timestamp Frame button (see page 5-19). When you change the timestamp display format, the time display is recomputed.

Display timestamp as Interval In interval mode, the time between trace frames is displayed. When you change the timestamp display format, the time display is recomputed.

You can choose to display timestamp in alternate modes in the Emulator Trace and Command windows.

Show preceding source comments This option, when enabled, shows up to 10 lines of source code that precede the line that generated the current code being displayed, if these lines did not generate any code themselves. With this option disabled, only the line that is indicated in the debugging information as generating the code is displayed in source interleave mode.

Whenever this option is changed, the display is cleared and redrawn with the new information.

Actions menu commands

Six options provide easy access to operations.

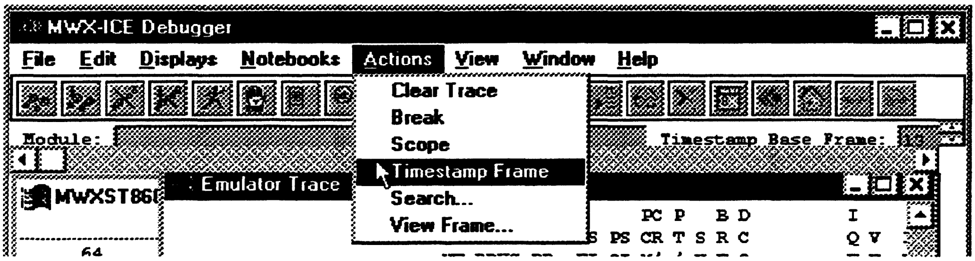


Figure 5-5 Emulator Trace window Actions menu

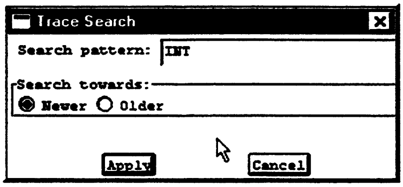
Clear Trace The Clear Trace option clears the emulator's trace buffer and updates the display. Note that a trace buffer contains two or three trace frames after it has been cleared. These trace frames are labeled Beginning of Trace and End of Trace.

Break The Break option attempts to set a breakpoint at the currently highlighted frame of trace. The frame must have a module name and a line number associated with it in the Module and Line status fields. Without a module or line number, a breakpoint cannot be set, and an appropriate error message is displayed.

Scope The Scope option makes the currently highlighted trace frame the scope, both for symbol accesses and in the Code window. The current trace line must have a module associated with it with it in the Module status field. If there is a source line number as well, this is used. Without a module or line number, the scope cannot be determined, and an appropriate error message is displayed.

Timestamp Frame This option makes the currently highlighted trace frame the base frame for offset timestamp measurement. It represents time zero, and all timing is relative to it. Selecting a new base frame clears the time values and displays the new timestamp measurements.

Search The Search option opens a dialog box that you use to perform a string search of the trace buffer.



Search pattern: INT

Search towards:
● Newer ○ Older

Figure 5-6 Trace Search dialog

➤ To enter a search pattern

1. Enter the exact string; the search is case-sensitive.
2. Select whether to search backward through the buffer (older) or forward (newer).
3. Click Apply.

   To stop a lengthy search, click Stop ✋ .

If a matching pattern is found, the frame containing the requested pattern is placed in the top line of the trace window. If the pattern is not found, a message appears.

To identify all instances of a pattern in the buffer, use the **tsrch** utility in the Command window.

Note

When searching for a data pattern or address, use the byte order as it would appear in a raw trace display with **drtdata** set to **all.**

View Frame The View Frame option opens a dialog box that prompts you for a frame number. If the value entered is a number, trace will be positioned to the requested frame. If the number is greater than the number of frames in trace, trace will be positioned to the oldest frame. If the value entered is not a number, an error box is displayed, and you are asked to enter a new value.

Trace display variables

Search Keywords: DRTDATA

There are three trace variables that provide additional control over elements of the trace display. You enter these commands in the Command window. For detailed information about these display variables, see Help.

Variable	Description
drtdata	Controls whether the DATA column in raw trace displays only the bytes that are valid in a given transfer (**drtdata notall**), or all four bytes that are on the data bus at the time (**drtdata all**).
drtfull	Controls whether all signals are displayed in raw trace.
dxlabels	Controls whether symbols are shown for branch destinations in disassembled trace. Use **dxlabels on** when you are using the MMU and address translation is *transparent,* or when the instruction MMU is not enabled. That is, use **dxlabels on** when the physical (effective) address is exactly same as the physical (real) address.

Emulator Trace status

When the Emulator Trace window is active, status boxes show the current trace mode being displayed, as well as other information about the trace buffer.

Module Displays the source module (if any) associated with the currently highlighted trace line.

Line Displays the source line number (if any) associated with the currently highlighted trace line.

Timestamp base frame Shows the frame number that is used as the starting point (time zero) when the offset format is selected.

Using the Command window trace display

Use the Command window as your trace interface when you need command-driven control of the trace system. Using keyboard commands, macros, or include files, you can

- Select the portion of the buffer to display.
- Change trace display configuration.
- Change timestamp format and offset base frame.
- Search for address, data, and status patterns in trace.
- Save trace to a file.
- Clear trace.

Note

See "Preparing to capture trace" on page 5-4 to set up the trace system for capturing trace.

Displaying raw trace

Raw trace always displays the trace frame number, address, data, timestamp, and certain signal type identifiers.

➤ To display raw trace

- Use the **drt** command:

Type	To do this...
`drt`	Display the most recent screen of trace.
`drt` *nnnn..mmmm*	Print the range of trace frames to the command window as a scrolling display from oldest to latest.
`drt` *nnnn*	Display a screen of trace containing the specified frame number.

The header for each page of trace identifies the address, data, and control signals displayed. See Help or "Raw trace display description" on page 5-35 for a description of display fields and symbols.

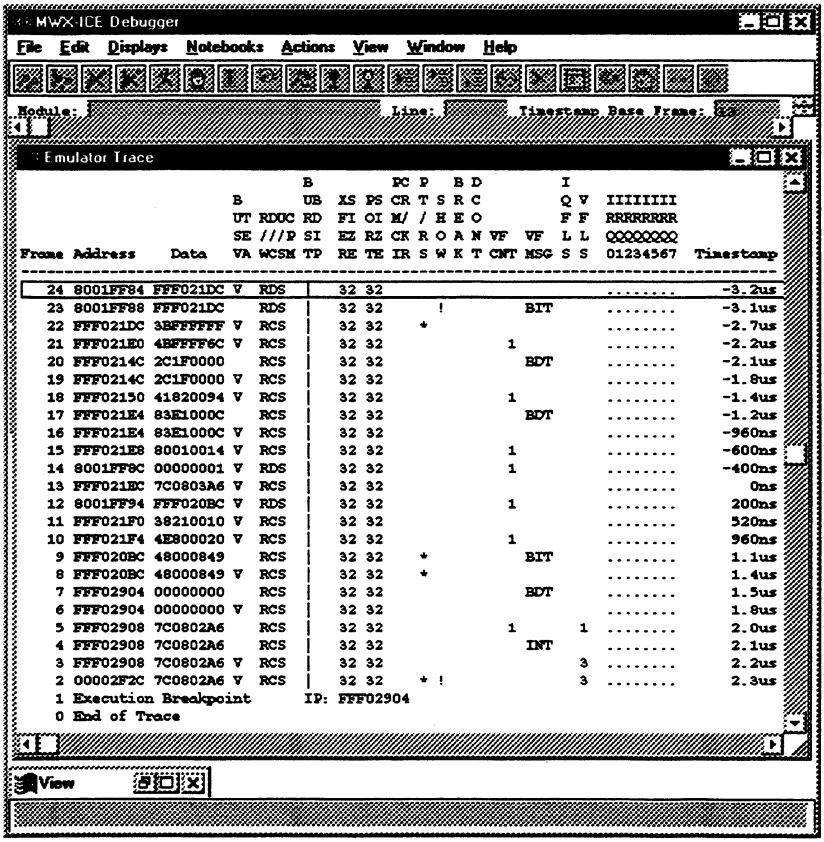
➤ To specify raw trace display fields

- Use these commands:

Type	To do this...
`drtfull`	Control whether all signals are displayed in raw trace.
`drtdata notall`	Display only the valid data bytes in a bus cycle.
`drtdata all`	Display all four bytes of the data bus.

Determining the number of trace frames

To determine the number of frames in the buffer, use the **trframes** command. The most recent frame is frame 0. The oldest frame is the value returned by the **trframes** command.



```
                                B         PC P   B D         I
                        B       UB  XS PS CR T S R C         Q V  IIIIIIII
                        UT RDUC RD  FI OI M/ / H E O         F F  RRRRRRRR
                        SE ///P SI  EZ RZ CK R O A N VF  VF  L L  QQQQQQQQ
Frame Address    Data   VA WCSM TP  RE TE IR S W K T CNT MSG S S  01234567  Timestamp
------------------------------------------------------------------------------------
   24 8001FF84 FFF021DC V  RDS  |   32 32                          ........    -3.2us
   23 8001FF88 FFF021DC    RDS  |   32 32      !         BIT      ........    -3.1us
   22 FFF021DC 3BFFFFFF V  RCS  |   32 32    *                    ........    -2.7us
   21 FFF021E0 4BFFFF6C V  RCS  |   32 32              1          ........    -2.2us
   20 FFF0214C 2C1F0000    RCS  |   32 32                BDT      ........    -2.1us
   19 FFF0214C 2C1F0000 V  RCS  |   32 32                         ........    -1.8us
   18 FFF02150 41820094 V  RCS  |   32 32              1          ........    -1.4us
   17 FFF021E4 83E1000C    RCS  |   32 32                BDT      ........    -1.2us
   16 FFF021E4 83E1000C V  RCS  |   32 32                         ........    -960ns
   15 FFF021E8 80010014 V  RCS  |   32 32              1          ........    -600ns
   14 8001FF8C 00000001 V  RDS  |   32 32              1          ........    -400ns
   13 FFF021EC 7C0803A6 V  RCS  |   32 32                         ........       0ns
   12 8001FF94 FFF020BC V  RDS  |   32 32              1          ........     200ns
   11 FFF021F0 38210010 V  RCS  |   32 32                         ........     520ns
   10 FFF021F4 4E800020 V  RCS  |   32 32              1          ........     960ns
    9 FFF020BC 48000849    RCS  |   32 32    *           BIT      ........     1.1us
    8 FFF020BC 48000849 V  RCS  |   32 32    *                    ........     1.4us
    7 FFF02904 00000000    RCS  |   32 32                BDT      ........     1.5us
    6 FFF02904 00000000 V  RCS  |   32 32                         ........     1.8us
    5 FFF02908 7C0802A6    RCS  |   32 32              1       1  ........     2.0us
    4 FFF02908 7C0802A6    RCS  |   32 32                INT      ........     2.1us
    3 FFF02908 7C0802A6 V  RCS  |   32 32                      3  ........     2.2us
    2 00002F2C 7C0802A6 V  RCS  |   32 32    * !               3  ........     2.3us
    1 Execution Breakpoint      IP: FFF02904
    0 End of Trace
```

Figure 5-7 Raw trace showing all signals (drtfull on /trqual dxqual).

Displaying all signals in raw trace

The **drtfull** command controls whether all the signals are displayed in raw trace. If you find the trace display to be too cluttered, you can suppress the display of some signals.

Search Keywords: Raw trace

The **drtfull** command does not determine what is captured. It simply selects whether to display all signals that have been captured in trace. The trace capture variable **trqual** controls which signals are captured. There are two basic groups of signals: the group of signals captured in clock mode, and the group of signals captured in dxqual and bus modes.

Once the trace is captured, you can use the drtfull option to switch back and forth between a full display of all signals, or a filtered display.

Limiting the data display

When fewer than 32 bits of data are accessed in a bus cycle, the most useful display is as the processor *sees* the data on the bus. The **drtdata** softswitch allows you to select whether to show only the data bytes active for the current bus cycle (**notall**) or all 32 bits (**all**), even if some bits were ignored by the processor.

To determine where a byte occurs so you can mask it in an event statement, simply switch from **notall to all**, and re-display the trace.

```
>DRTDATA NOTALL
Frame Address   Data
0472 00100008     4FF9
0471 0010000A     0F00
0470 0010000C FFFC41F9
```

```
>DRTDATA ALL
Frame Address   Data
0472 00100008 4FF90F00
0471 0010000A 4FF90F00
0470 0010000C FFFC41F9
```

Excluding non-bus frames from raw trace

You can restrict *raw* trace to bus cycles. Use the **trqual bus** command to exclude non-bus cycles during capture.

Configuring the timestamp display

Search Keywords: Timestamp

You can use the emulator's timestamp to perform a variety of timing measurements automatically during run and to display those values in raw trace. There are two timestamp modes:

- Interval: times the intervals between cycles.
- Offset: sets time relative to a specified frame in trace.

To configure the timestamp information, use the following commands:

Set	To do this...
`trstamp interval`	Configure display to show time between cycles.
—or—	
`trstamp offset`	Configure display to show time relative to trace frame specified by **trbase**.
`trbase` *`nnnn`*	Trace frame that is time zero.

For example, to see the intervals between interrupt 5 requests, enter the following commands:

```
trace off
trinit off
when status==irq5 then trone
```

Then open the Emulator Trace window and select Interval as the Timestamp format.

Displaying disassembled trace

Search Keywords: Disassembled trace

Disassembled trace can be displayed as assembly instructions, C-source instructions, or source, assembly and raw trace interleaved. Several variables control what is displayed, and three commands enable navigation within the display.

Note

Disassembled display is possible only if you have used the proper setup prior to trace capture. This must include setting **trqual** to **dxqual**. See “Preparing to capture trace” on page 5-4 for procedures.

➤ To display disassembled trace

- Use these commands:

Type	To do this...
`dt` *`number`*	Display most recent screen of trace or specified range.
`dtb`	Display trace going backward (from current frame toward earliest).
`dtf`	Display trace going forward (from current frame toward most recent).

➤ To select what is displayed

- Use these commands:

Type	To do this...
`trdisp` *`type`*	Display assembly (asm), source (src), or both (both).
`dxinsert on`	Interleave raw trace (optional).
`drtfull`	Show all signals available (on), or filter the display (off).

Figure 5-8 shows a typical example of disassembled trace that includes interleaved raw trace. For an explanation of each raw trace field, see page 5-35.

➤ To display symbols for branch destinations

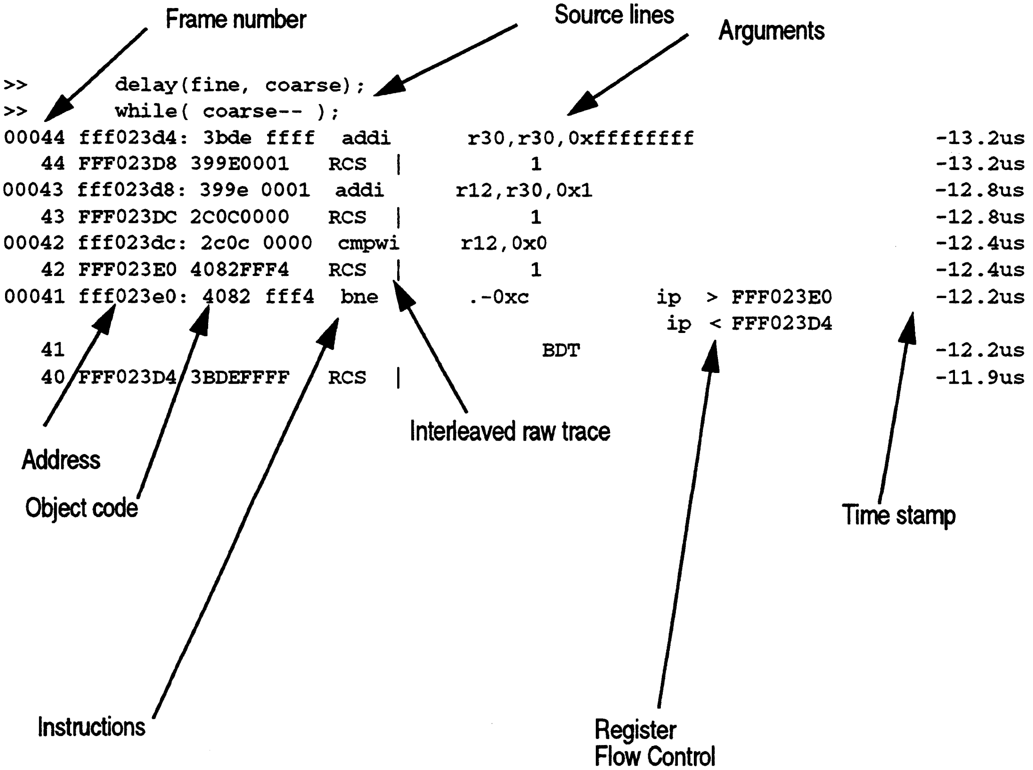
- Use this command:

Type	To do this...
`dxlabels on`	Display symbols in branch destinations (on), or show relative address (off). The disassembler looks in the symbol table for branch destination (MMU must translate 1:1, or be disabled).

➤ To include raw trace cycles

- Use this command:

Type	To do this...
`dxinsert on`	Include relevant raw trace cycles in the disassembled display.



```
>>        delay(fine, coarse);
>>        while( coarse-- );
00044 fff023d4: 3bde ffff  addi      r30,r30,0xffffffff                 -13.2us
   44 FFF023D8 399E0001   RCS  |       1                                -13.2us
00043 fff023d8: 399e 0001  addi     r12,r30,0x1                         -12.8us
   43 FFF023DC 2C0C0000   RCS  |       1                                -12.8us
00042 fff023dc: 2c0c 0000  cmpwi    r12,0x0                             -12.4us
   42 FFF023E0 4082FFF4   RCS  |       1                                -12.4us
00041 fff023e0: 4082 fff4  bne      .-0xc         ip  > FFF023E0        -12.2us
                                                  ip  < FFF023D4
   41                                BDT                                -12.2us
   40 FFF023D4 3BDEFFFF   RCS  |                                        -11.9us
```

Figure 5-8 Source and assembly display of disassembled trace interleaved with raw trace

Searching for patterns in Command window trace

The **tsrch** command is a command-line utility for searching trace. Unlike the Search facility in the Emulator Trace window, **tsrch** is capable of locating and indexing multiple instances of a pattern in trace.

You can specify a range of frames for the search or use the default to search the entire buffer. Search patterns are entered as address, data, or status, or combinations of these elements. Output reports the trace frame numbers and shows all frames with the matching pattern in **raw trace** format. Using **drt** or **dt** *frame_number*, you can display each frame in the context of its execution history.

Search Keywords:
TSRCH
Trace, searching

For example, to search the entire buffer for all reads of the data value 0xC453 at address 0x789f, enter the following:

```
tsrch addr==0x789f && data==0xC453 && status==rd
```

The address and data *values* can be a simple value (0x1000) or a value with a "care" mask (0x1000&=0xf000). Both can be expressed as an equivalency (=) or as an inequality (!=).

Search Keywords:
Status
mnemonics

Status comparator values are entered mnemonically using the same mnemonics recognized by the event system. Mnemonics can be logically ANDed using the vertical bar (**|**): status=*mnemonic* | *mnemonic*. See the Help description of **tsrch** for valid status mnemonics.

Saving trace to a file

Trace displayed in the Command window can be printed to file and reviewed.

The **journal and log** commands (or the associated Debugger Files notebook pages) let you easily store trace history in a file. You can even create this file *without* stopping emulation. After trace is saved, you can edit the file to add comments for future reference. For example, comments may be added to aid in documenting failure conditions.

- ❑ Log files are formatted such that the commands issued to the debugger can be re-run by loading the file. Data printed to the command window, such as trace, are recorded as comments.
- ❑ Journal files present trace in a form that closely approximates actual screen display. Journals record your command input, the data printed to the Command window, and any error or warning messages.

You can use either the **log** or **journal** commands or the guided configuration provided by the Debugger Files notebook. Select the “File Loading/Execution” topic in the main contents window of Help for procedures.

Clearing the trace buffer

➤ To clear the trace buffer

- ■ Use the **trclr** command.

Notes on using trace

This section includes a variety of topics that apply to capturing and displaying trace in the Emulator Trace window, and in the Command windows.

Logical addressing

If the target uses the MMU of the MPC8XX, the disassembler can use trace information to display addresses as long as the logical (effective) and physical (real) addresses are the same. In other words, if address translation is *transparent*, the address and symbol information shown in trace will be correct.

If you are using the MMU, and your address translation is transparent, you should set an emulator trace variable (**dxlabels on**). The **dxlabels** variable controls whether symbol table lookups are performed. As long as the MMU doesn't change the address, the correct symbols will be shown for branch destinations.

If you use the MMU and the logical and physical addressing is identical, keep **dxlabels** on.

If you are not using the MMU at all, the default (**dxlabels on**) will still be applied. But if you turn **dxlabels off,** and then later disable the MMU, you must turn **dxlabels on** again.

Trace compression

When **trqual** is set to **dxqual**, the SuperTAP's trace system uses two ways of reducing the number of trace frames produced by executing a given segment of code. This allows longer code segments to be traced without loss of information. This compression of the trace is done by synthesizing a couple of artificial signals from the processor's raw signals.

The VF CNT signal is synthesized by simply counting the number of successive *sequential instruction* messages received on the VF pins, to a maximum of 15 instructions. This number is reset whenever any other VF message appears, and every time a trace frame is written.

Additional saving of trace frames is achieved with the IQFLS signal. As outlined in Section 18.1.1.1 of the MPC860 User's Manual, when instructions are flushed from the processor's prefetch queue, the number of flushed instructions is reported on the next clock cycle after the fetch report of the instruction that caused the flush. To avoid writing two trace frames in this case, the trace system combines them into one frame by saving the VF output from the first frame as the VF MSG and the second as the IQFLS. A complication of combining the two trace frames occurs when the first frame must be written anyway because a valid bus transfer occurred on that clock. In this case, the trace system writes both frames, repeating the VF MSG on the second frame. For this reason reports of the VF=4–7 messages (such as BIT and BDT) are sometimes repeated in a second trace frame for a single branch.

Using these techniques, we only have to save trace frames in **dxqual** mode when a valid bus cycle (including show cycles) occurs, when the sequential instruction count reaches 15 and rolls over, or when a non-sequential VF report is made. All other clock cycles are discarded in this trace mode.

A consequence of this compression is that many instructions may be reported on a single trace frame, with a single timestamp. The timestamp corresponds to the time of the final clock cycle for this series of instructions. Keep in mind that for all instructions on the MPC8XX processor, the time reported shows when the instruction was fetched; the exact time that the instruction executed cannot be determined in this architecture.

Some common problems disassembling trace

The emulator captures the processor activity on each bus cycle. From this raw data, the emulator is usually able to reconstruct the assembly instructions. However, there are times when the disassembler may be unable to reconstruct instructions from trace memory. The following sections describe some conditions when this could occur.

Unable to disassemble trace frames

The following are the most common reasons that the trace disassembler is unable to complete disassembly:

- ❑ The **trqual** variable is set to **bus** or **clock**.
- ❑ The **trqual** variable is changed from an unsupported mode (**bus**, **clock**) to the supported mode (**dxqual**) without clearing the trace buffer.
- ❑ Trace was collected while processor show cycles were disabled (**showinst none**).
- ❑ Clock-doubling is turned on. Clock-doubling is enabled by setting bits EBDF of register SCCR to 01 [see the Motorola MPC860 User's Manual, Section 5.8].
- ❑ Visibility functions are disabled (bits DBGC of register SIUMCR are *not* set to 11 [Motorola UM, Section 12.4.1.1]).
- ❑ The event system setup turned trace on and off before acquiring sufficient information for disassembly. No instructions can be displayed. This also occurs with peek/ poke trace (**ppt**). However, you can still display raw trace (**drt**).
- ❑ The application code runs a long loop before breaking, and **showinst** is set to **indirect** and the instruction cache is enabled. In this case the direct branch instruction is cached, and the disassembler may not be able to reconstruct execution. Setting **showinst** to **flow** may correct this

problem, by enabling show cycles for all direct and indirect branching. In cases like this, there is no performance penalty. While there is usually a loss of CPU performance if **flow** is selected, in this case there is no performance degradation.

Disassembled trace display description

C source code

When displaying both C source and assembly instructions, C source lines begin with >> and precede the assembly instructions associated with them.

Assembly instructions

Assembly instructions are decoded and displayed as follows:

Column	Description
Frame number	An index of the bus cycle in the trace buffer. The most recently traced cycle is the lowest number.
Address	Address of instruction in memory.
Object code	Numeric representation of assembly code.
Instruction	Processor instruction and operand.Where possible, on branch and trap instructions, the simplified Motorola PowerPC mnemonics are used.
Arguments	Instruction arguments.
Flow Control	Shows non-sequential changes to the IP. For each change, the old IP is shown as *IP > address*. The new IP is shown as *IP < address*.

Timestamp Timestamps only appear for trace frames representing a bus cycle. If the timestamp format is interval (**trdisp interval**), the timestamp information is recorded as the interval between successive bus cycles. If the timestamp format is offset (**trdisp offset**), the timsestamp is shown relative to the specified trace frame number (**trbase**).

If symbol information can't be found

If symbol information can't be found, the cause may be MMU translation. If the MMU changes the address during translation, only the address will be displayed in trace, and not the symbol. For example, the following example shows a branch instruction to a relative address.

```
b .+084c
```

As long a the MMU doesn't change the address during translation, the symbol information should be available and will appear in trace.

You also need to make sure the trace option **dxlabels** is on. This option tells the disassembler to check the symbol table for branch destinations.

Note that if the MMU changes the address during translation (that is, if the logical to physical address does not correspond 1:1), the disassembler may display incorrect symbol information. In this case, make sure that **dxlabels** is off.

Flushed instructions

Instructions that are fetched, but not executed--or only partially executed--are indicated by the FLSH mark. The FLSH appears just before the timestamp in the trace disassembly.

Invalid instructions

An instruction followed by two exclamation points (!!) indicates that instruction is invalid. An invalid instruction is always followed by an interrupt. For example, a PowerPC floating point instruction would be invalid because floating point instructions are not implemented in the MPC860.

```
stfdu!! r10, 0x000a(r10)
```

A common cause for an invalid instruction is a bug in the application code that causes the processor to try to execute code in a section of memory where there aren't any instructions. In this case, the "instruction" fetched would not likely contain a valid opcode.

Questionable instructions

An instruction followed by two question marks (??) indicates that the instruction may or may not have been architecturally executed. In other words, it's status is questionable. Questionable instructions only appear shortly before a discontinuity in trace. This discontinuity occurred before the disassembler could determine if the instruction was executed or not. All the disassembler knows is that it was fetched, but it's status is unknown.

```
00021 00080000: 3d40 1234  addis     r10,0,0x1234
00020 00080004: 394a 5678  addi      r10,r10,0x5678
00019 00080008: 3d60 acac  addis??   r11,0,0xffffacac
00018 0008000c: 396b fefe  addi??    r11,r11,0xfffffefe
00017 00080010: 3d80 e1e1  addis??   r12,0,0xffffe1e1
00016 00080014: 398c beef  addi??    r12,r12,0xffffbeef
00015 00080018: 3de0 0007  addis??   r15,0,0x7
-------End of Contiguous Section of Qualified Trace----
```

Raw trace display description

The following table describes the raw trace fields that can be captured and displayed. The type of trace captured depends upon the setting of the **trqual** command. The **drtfull** command filters the captured trace data for display purposes. For information on setting the **trqual** options and displaying the raw trace signals, see Help.

Column	Description
FRAME	The decimal count of the line in the trace buffer. Line 0 corresponds to the most recently traced cycle.
ADDRESS	The hex value of the address bus.
DATA	The hex value of the data bus.
R/W	R read
	W write
XFER SIZE	The transfer size in bits (8, 16, or 32), translated from the TSIZ0:1 codes.
STS	S Special transfer start (STS*) asserted.
TS	T Transfer start (TS*) asserted.
AS	@ Address strobe (AS*)
TA	A Transfer acknowledge (TA) asserted.
TEA	E Transfer error acknowledge (TEA) asserted.
BI	> Burst inhibit (BI*) asserted.
BURST	Burst transaction (BURST*)
	$ Asserted
	| Negated
BDIP	< Burst data in progress (BDIP*).
BR	R Bus request (BR*) asserted.
BG	G Bus grant (BG*) asserted.

Column	Description
BB	B Bus busy (BB*) asserted.
IRQ0, 1, 7	These signals are multiplexed with LSA0, LSA1, LSA7. See the **siga_mux** command. • negated *n* asserted (where *n*=0, 1, 7)
IRQ2:6	These signals are multiplexed with LSA2:6, RSV*/CR/KR, WP, PCMCIA_B, DPx. See the **sigb_mux** command. • negated *n* asserted (where *n*=2, 3, 4, 5, 6).
LSA 0, 1, 7	Logic State Analysis Signal (LSA0, 1, 7). These signals are multiplexed with IRQ0, IRQ1, IRQ7. See the **siga_mux** command. Use this option when you have an LSA probe connected to your target system. + The LSA signal is a physical "1." - The LSA signal is a physical "0."
LSA2:6	Logic State Analysis Signal (LSA2:6). These signals are multiplexed with IRQ2:6, RSV*/CR/KR, WP, PCMCIA_B, DPx. See the **sigb_mux** command. Use this option when you have an LSA probe connected to your target system. + The LSA signal is a physical "1." - The LSA signal is a physical "0."
DPn	Data parity (DP0:3). These signals are multiplexed with IRQ2:6, RSV*/CR/KR, WP, PCMCIA_B, LSA2:6. See the **sigb_mux** command. + The DP signal is a physical "1." - The DP signal is a physical "0."

Column	Description
IWPn	Instruction watchpoint signals (IW0:2). + The IW signal is a physical "1." - The IW signal is a physical "0."
LWPn	Load/Store watchpoint signals(LW0:1) + The LWP signal is a physical "1." - The LWP signal is a physical "0."
CHIP SEL	Chip select lines 0:7. • negated. *n* asserted (where *n*=0:7).
WEn	W Write enable asserted (WE0:3*).
BS	S Byte selects asserted (BS_B0:3, BS_A).
GPLA0:3, 5	General purpose lines (GPL_A0:3, GPL_A5) • negated. *n* asserted (where *n*=0:3, 5).
UAGA$	UPWAITA/GPLA4* + This multiplexed signal is a physical "1." - The multiplexed signal is a physical "0."
UBGB4	UPWAITB/GPLB4* + This multiplexed signal is a physical "1." - The multiplexed signal is a physical "0."
RSV	Reservation (RSV) This signal is multiplexed with LSA2:6, WP, PCMCIA_B, DPx. See the **sigb_mux** command. R asserted.

Column	Description
CR/KR	X Either Cancel reservation asserted (CR*), or Kill reservation asserted (KR*), which means in either case that a reservation has been cancelled.
BUSV	V Address bus, data bus, and transfer attributes are valid. The SuperTAP synthesizes this signal. Only displayed when **trqual** is set for **bus** or **dxqual**, and **drtfull** is **on**.
D/C	D Data.
	C Code.
U/S	U User.
	S Supervisor.
CPM	# The bus is driven by the MPC8XX Communications Processor module (CPM) instead of the Core.
PCMCI	P The bus is driven by the MPC8XX PCMCIA adapter module.
PT/RS	Program trace or Reservation depending upon the value of the D/C column in trace.
	* Program trace if D/C column is code (C).
	* Reservation if D/C column is data (D).
VF CNT	The SuperTAP creates this signal which tracks sequential instructions only. It is the count of the number of consecutive sequential instructions reported on this frame.

<table>
<tr><th>Column</th><th>Description</th></tr>
<tr><td>VF MSG</td><td>Reports the value of the VF pins [0..2] on the current clock cycle.
<table>
<tr><th>Message</th><th>VF</th><th>Meaning</th></tr>
<tr><td></td><td>0</td><td>None</td></tr>
<tr><td>SEQ</td><td>1</td><td>Sequential</td></tr>
<tr><td>BNT</td><td>2</td><td>Branch not taken</td></tr>
<tr><td>VSNC</td><td>3</td><td>VSYNC signal (disabled in hardware)</td></tr>
<tr><td>INT</td><td>4</td><td>Interrupt</td></tr>
<tr><td>BIT</td><td>5</td><td>Branch indirect taken</td></tr>
<tr><td>BDT</td><td>6</td><td>Branch direct taken</td></tr>
<tr><td>BNTF</td><td>7</td><td>Branch not taken, flush follows</td></tr>
</table></td></tr>
<tr><td>IQFLS</td><td>The value of the VF pins [0..2] on a clock where no instruction fetches occurred. This is when the preceding VF had the value 4-7. IQFLS shows the number of instructions flushed from the "prefetch" queue.</td></tr>
<tr><td>VFLS</td><td>The value of the VFLS pins [0..1]. Shows the number of instructions flushed from the history buffer on the current clock cycle. A value of 3 indicates that the emulator is paused (the processor is in Debug mode).</td></tr>
<tr><td>DCONT</td><td>D Discontinuous trace frames.</td></tr>
<tr><td>BREAK</td><td>B Asynchronous break request.</td></tr>
<tr><td>Sft Rst</td><td>Rst SRESET* asserted.</td></tr>
<tr><td>TIMESTAMP</td><td>If the timestamp format is interval (<b>trdisp interval</b>), the timestamp information is recorded as the interval between successive bus cycles. If the timestamp format is offset (<b>trdisp offset</b>), the timsestamp is shown relative to the specified trace frame number (<b>trbase</b>).</td></tr>
</table>

Why do some of the trace frames appear empty?

When **drtfull** is **off** and **trqual** is set to either **bus** or **dxqual**, you may see trace frames that appear to be empty except for the trace frame number and timestamp. These *empty* frames appear when the bus valid signal is not asserted (BUSV). In these frames, the only signals that are valid are DCONT and BREAK, the visibility signals (VF CNT, VF MSG, IQFLS), and the timestamp. All of the other signals (including address and data) may not show valid values, and their display is suppressed. The pins for these signals are not actively driven at these times. When **drtfull** is **off**, the display of the visibility signals is also suppressed.

When **drtfull** is **on**, you can see that the *empty* trace frames contain valid visibility information (VF CNT, VF MSG, IQFLS).

Chapter 6
Using Basic Breakpoints

This chapter covers the breakpoint features of MWX-ICE. Basic breakpoints are tools for interrupting emulation for insight into code execution and target function. Breakpoints interrupt emulation after memory accesses or before executing an instruction.

Contents **Page**

How can you use breakpoints

Once you have run your program and discovered a problem, the next step is typically to decide where to break program execution so that you can find the source of the problem.

You use a breakpoint to examine behavior of the target under certain controlled conditions. This is very helpful in isolating bugs when troubleshooting hardware and software in the target environment.

Use the event system to monitor compound conditions and perform various emulator actions based on those conditions. The same emulator resources are used to support both breakpoints and the event system. See the section “Working within the limits” for each breakpoint type for information on resources.

Breakpoint types

You can set two types of breakpoints: access breakpoints and instruction breakpoints. In addition, you can specify how the emulator and on-chip resources are used to implement breakpoints. However, in most cases it is best to let the SuperTAP manage the breakpoint resources (**bptype choose**).

Access breakpoints

Access breakpoints break on reads or writes to data and program locations. You can limit an access breakpoint to break exclusively on read or write accesses by using the applicable command (**breakread**, **breakwrite**).

An access breakpoint is set for a single address or an address range. You can use up to 10 single-address access breakpoints or five ranges, or some combination of both types. You can set up to 10 access breakpoints when the **bptype** is set to **choose**. This is the default setting.

On-chip access breakpoints There are two on-chip access breakpoints available. When you select on-chip breakpoints (**bptype onchip**), the breakpoint occurs immediately after the data is accessed. Unlike hardware access breakpoints, on-chip access breakpoints only break on data accesses, not on instruction accesses.

Hardware access breakpoints You can set up to eight emulator hardware access breakpoints (**bptype hw**). When you select hardware breakpoints, the breakpoint actually occurs some cycles after the data is accessed. This is known as *skid*. However, it is easy to locate the exact bus or clock cycle where the break occurred when you examine raw trace. The trace frame where the breakpoint occurred is marked with a B. Note that when you explicitly select hardware breakpoints (**bptype hw**), instruction breakpoints (**bi**) and temporary breakpoints (go *address*), and step commands do not work. Again, in most cases it is best to let the SuperTAP manage the breakpoint resources (**bptype choose**).

Instruction breakpoints

Instruction breakpoints break just before executing the selected instruction.

When **bptype** is set to **choose** or **sw**, up to 50 instruction breakpoints can be set in writable memory. Instruction breakpoints work by replacing the actual instruction in memory with a illegal instruction opcode.

When **bptype** is set to **onchip**, you can set four instruction breakpoints. These on-chip instruction breakpoints can be set in ROM or RAM. When **bptype** is set to **choose**, the SuperTAP automatically uses on-chip breakpoints for instructions in ROM.

When **bptype** is set to **hw**, instruction breakpoints and temporary breakpoints (**go** *address*) do not work. Note that the step commands also use temporary breakpoints.

Basic breakpoint commands

Control breakpoints with the following commands. For complete command descriptions see Help.

To	Use	Abbreviation
Set a breakpoint to break before execution of a specified instruction or range.	breakinstruction	bi
Select among hardware, software, or on-chip instruction breakpoints.	bptype	none
Set a breakpoint on access to specified address or range.	breakaccess	ba
Set a breakpoint on a read at specified address or range.	breakread	br
Set a breakpoint on a write at specified address or range.	breakwrite	bw
Clear all breakpoints or a breakpoint by number.	clear	cl

Setting breakpoints

See the sections “Access breakpoints (ba, br, bw)” on page 6-7 and “Instruction breakpoints (bi)” on page 6-12 for information on using the **ba**, **br**, **bw**, and **bi** commands.

You can set breakpoints by entering commands in the Command window, or by using the Breakpoint page of the Execution Controls Notebook, or by using the BreakI button.

Displaying breakpoints

The debugger assigns a number to each breakpoint for reference and displays them in the Breakpoints window. If the breakpoint numbers are not displayed select “Show Break #” from the View menu.

Clearing breakpoints

Breakpoints may be cleared using the **clear (cl)** command. You can clear an individual breakpoint by giving its number or clear all breakpoints by not specifying a number.

Example	Description
`cl 3..5`	Clears breakpoints numbered 3 through 5.
`cl 4`	Clears breakpoint number 4.
`cl`	Clears all breakpoints.

Breakpoints can also be cleared with the Clear button: double-click the breakpoint and choose the Clear button.

Attaching macros to basic breakpoints

The **breakaccess**, **breakread**, **breakwrite**, and **breakinstruction** commands can all be set to invoke a macro when program execution is broken. For example, to invoke *my_macro* after a breakread at line #20, enter

```
br #20 ;my_macro()
```

Access breakpoints (ba, br, bw)

Access breakpoints break on reads or writes to data and program locations. You can limit an access breakpoint to break exclusively on read or write accesses by using the applicable command.

You can use up to 10 single-address access breakpoints or five ranges, or some combination of both types. You can set up to 10 access breakpoints when the **bptype** is set to **choose.** This is the default setting.

Setting an access breakpoint

➤ To select memory access attributes

■ Set the **address** command for the addressing mode appropriate for your target. For example,

```
address access physical
```

For settings, see Help for the **address** command.

➤ To enter breakpoint commands

■ For access breakpoints use the following syntax:

```
{ba | br | bw} address | address_range [;macro_name()]
```

Determine which type of access breakpoint you need, the address you want to break on, and the name of any macro you want to invoke when program execution is broken.

Enter the breakpoint command on the command line. For example,

```
br 20h..30h ; foo()
```

Search Keywords: Breakpoints window

Breakpoints can also be set using the Breakpoint page of the Execution Controls Notebook.

Address values

Single addresses and address range expressions can contain actual memory locations, symbols, constants, line numbers, and operators. You can specify an address range by separating two addresses with two periods (*address1..address2*). A byte offset can also be specified for a range using *+n*. For more information on expressions and symbolic referencing, see Help.

Memory qualifiers

The settings for the **address** command affect how addresses are qualified for access breakpoints. The settings for these commands for the **access** memory type determine the addressing mode for a breakpoint address. Note that when **bptype** is set for **onchip**, the **ba**, **br**, and **bw** commands always use logical addressing.

For syntax, see the "Memory and File Handling" command groupings in Chapter 9; for detailed information, see Help for the **address** command.

Examples

Example	Description
`BR 0x300`	Sets a read access breakpoint at address 300 (hexadecimal).
`BW @CDEMON\\led_port`	Sets a write access breakpoint at the address of the array led_port in the root named @CDEMON.
`BA flags..flags+10`	Sets read/write access breakpoint starting at the address of the array flags and ending 10 bytes after the address of flags.

Example	Description
`BR 20h..30h;FOO()`	Sets read access breakpoints from address 20h (hexadecimal) to 30h and executes the macro FOO on every breakpoint between these addresses.
`BR &flags[0]`	Sets a read access breakpoint at the address of array element flags[0].

`BA &count;`
`when(k<30)`
	Sets a read/write access breakpoint at the address of count and only stops when the macro evaluates k to be less than 30.

`BA prime`	Sets a read/write access breakpoint at the address referred to by the value in variable prime. This command is correct if prime is a pointer. The breakpoint is set at the value of the variable prime. For example, if the value of prime is 0x0123, a breakpoint is set at the address 0x0123. This command may not be correct if prime is a scalar, since the value in prime is treated as an address and the breakpoint is set at that address rather than at the address of the variable prime.

Example	Description
`BW &prime`	Sets a write access breakpoint at the address of the variable prime regardless of its type. This command is correct if prime is a scalar; it sets a breakpoint at the address of the variable prime. If prime is a pointer, the breakpoint is set at the address of the pointer rather than at the address it is pointing to (i.e., prime).

What happens when an access break occurs

An access breakpoint stops execution after the access occurs. Depending upon the type of access (instruction or data), the breakpoint will occur either immediately after the access, or some cycles past it. This discrepancy is a function of how the breakpoint is implemented. When **bptype** is set to **choose**, the emulator allocates the breakpoint resources (on-chip and in emulator hardware). In some cases, there may be a skid.

- When **bptype** is set for **onchip**, execution stops immediately following the access, so there is no skid. The SuperTAP automatically uses this type of breakpoint when the access is for data.
- When **bptype** is set for **hw**, the breakpoint skids. But you can still look in raw trace to locate the exact access that caused the break. This is the first trace frame where the break bit (B) is set. The SuperTAP automatically uses this type when the access is on an instruction, or when on-chip resources are depleted.

MWX-ICE performs the following functions when it encounters an access breakpoint:

1. Completes execution of the instruction at that location.
2. Suspends program execution.
3. Executes a macro (if one was specified when the breakpoint was set). Depending on the macro, the debugger will do one of the following:

   If the macro return value is true (nonzero), the debugger resumes execution at the instruction immediately after the breakpoint.

   If the macro return value is false (zero), the debugger returns to command mode and displays breakpoint information.
4. If a macro was not specified, MWX-ICE returns to command mode and displays updated breakpoint information.

Working within the limits

Access breakpoints consume system resources. In general, the emulator manages these resources and warns you when it makes adjustments and presents an error when resources are exhausted or when you attempt something that creates a conflict. So you need not concern yourself with more than the following general guidelines.

- ❑ Up to 10 access breakpoints are possible (8 hardware, 2 on-chip). You can have ten single-address breakpoints or five ranges, or some combination of both types.
- ❑ Use of access breakpoints reduces the resources available for the event system.
- ❑ On-chip access breakpoints only work for data accesses.
- ❑ Hardware access breakpoints have significant skid.

On-chip caches

When the processor on-chip instruction and data caches are used, you must make sure that instruction or data show cycles enabled, or the instruction/data caches will cause the access breakpoints to be missed when cache hits occur. If copy back caching is enabled for the data cache, access breakpoints may break in sections of code having nothing to do with the access; it is just where the processor decided to copy back the cached data to memory. In MWX-ICE, you can use the **showinst** command to enable instruction show cycles (**showinst indirect**). The data show cycles are enabled by setting the DSHW bit in the processor's SIUMCR register.

Instruction breakpoints (bi)

When you want to stop program execution on a particular instruction in your code, you use an instruction breakpoint (**bi**). With instruction breakpoints, the break initiates and completes before the instruction at the specified address is executed.

Up to 50 instruction breakpoints can be set (4 on-chip, and 46 software instruction breakpoints). They can be set from the command line or a one-time *temporary* breakpoint can be attached to the current MWX-ICE **go** instruction. The step commands also use temporary breakpoints.

Setting an instruction breakpoint

Some setup is necessary before initial use of instruction breakpoints and temporary breakpoints.

➤ **To perform initial setup**

1. Set the **address** command for the addressing mode appropriate for your target. For example,

   ```
   address exec physical
   ```

   For valid settings see the Help for the **address** command.
2. Make sure that **bptype** is set to **choose** or **onchip**.

If your code resides in ROM You can set up to four access breakpoints in ROM. If you have already used up the four on-chip breakpoints, you must map the ROM to emulator overlay memory. See "Copying memory contents between target and overlay" on page 3-16. In emulator overlay ROM, the area remains protected from target writes during program execution, but can be modified by MWX-ICE for breakpoint operations.

➤ To enter breakpoint commands

- For instruction breakpoints, use the following syntax:

  `bi` *`address`* `|` *`address_range`* `[;`*`macro_name`*`()]`

Determine the address you want to break on. Enter the breakpoint command on the command line. For example,

```
bi step ;when (i=3)
```

You can also set instruction breakpoints from the Breakpoint page of the Execution Controls Notebook, with the BreakI button, and by using the mouse shortcuts in the Code window.

Address values

Single addresses and address range expressions can contain actual memory locations, symbols, constants, line numbers, and operators. You can specify an address range by separating two addresses with two periods (*address1..address2*). A byte offset can also be specified for a range using +*n*. For more information on expressions and symbolic referencing, see Help.

- ❑ Ensure the address falls on an instruction boundary.
- ❑ Ensure the instruction resides in writable memory.
- ❑ Ensure that **bptype** is set to choose.

Setting temporary breakpoints

A temporary or one-time breakpoint is an instruction breakpoint attached to the current MWX-ICE **go** instruction. Temporary breakpoints are commonly used to skip over a section of code or a subroutine.

➤ **To set one-time breakpoints**

1. Perform the initial setup as described on page 6-12.
2. Determine a valid address you want to break on and the name of any macro you want to invoke when program execution is broken. Enter the **go** command. For example,

   ```
   go 0x1234
   ```

   Temporary breakpoints can also be set with the Go Until button or with options on the Go To notebook page.

Memory qualifiers

The settings for the **address** command affects how addresses are qualified for access breakpoints. The settings for these commands for the **exec** memory type determine the addressing mode used for a breakpoint address.

For syntax, see the "Memory and File Handling" command groupings in Chapter 9; for detailed information, see Help for **address**.

Examples

Example	Description
`BI #20`	Sets a breakpoint at line number 20. Source-level mode only.
`BI 0x2210..0x2216`	Sets breakpoints starting at address 2210 and ending at address 2216 (hexadecimal), assembly-level mode only.

Example	Description
`BREAKI #1..#4`	Sets breakpoints starting at line number 1 and ending at line number 4. May require module name.
`BI SIEVE\#28`	Sets a breakpoint at line number 28 in the module SIEVE.
`BI #15..#18;FOO()`	Sets breakpoints starting at line number 15 and ending at line number 18. Executes macro FOO after each line.
`BI #10;when(i==3)`	Sets a breakpoint at line number 10 and stops only if variable i is equal to 3.
`BI 0x93`	Sets a breakpoint at address 93 (hexadecimal), assembly-level mode only.
`BI step`	Sets a breakpoint at the address of step.

What happens when a software instruction break occurs

When a software instruction breakpoint is set, the specified target instruction is replaced with an illegal instruction; therefore, they can be set only in writable memory.

MWX-ICE performs the following functions when it encounters an instruction breakpoint:

1. Suspends program execution before the instruction at the breakpoint address is executed.
2. Replaces the illegal instruction with the original instruction at the breakpoint address.
3. Executes a macro (if one was specified when the breakpoint was set). Depending on the macro, the debugger will do one of the following:

   If the macro return value is true (nonzero), the debugger resumes execution starting at the instruction where the break occurred and displays break information.

   If the macro return value is false (zero), the debugger returns to command mode without executing the instruction where the break occurred.
4. If a macro was not specified, MWX-ICE returns to command mode without executing the instruction where the break occurred.

Software breakpoints and trace.

Trace memory shows the processor fetches and execution from the address a software instruction breakpoint is set to. It is the illegal instruction that is fetched and executed, not the instruction in your code. Also, the raw trace display may show extra fetches that result from filling the CPU pipeline. The break has occurred at the specified point, as is shown by the instruction pointer.

Working within the limits

Instruction breakpoints consume system resources. In general, the emulator manages these resources, warns you when it makes adjustments, and presents an error when resources are exhausted or when you attempt something that creates a conflict. So you need not concern yourself with more than the following general guidelines.

- The system can manage up to 10 access breakpoints (**ba**, **br**, **bw**). System resource are shared with the event system so the actual number of breakpoints available may vary.
- An instruction breakpoint resource is used for each instruction in an address range.
- Software breakpoints must occur in writable memory, either target RAM or emulator overlay RAM or ROM.
- Instruction breakpoint addresses must fall on instruction boundaries. Misaligned addresses will cause code corruption. Use the **printsymbols** or **disassemble** commands to evaluate instruction addresses.
- If you are setting instruction breakpoints or temporary breakpoints, the **bptype** option must be set to **choose**.
- There are four on-chip instruction breakpoints. The on-chip breakpoints can be set in target ROM.

Chapter 7

Using the Event System

This chapter covers the conditional control features of MWX-ICE when used with the SuperTAP event system.

The basic breakpoints feature and the event system can both be used to control emulation for insight into code execution and target function. Compared to basic breakpoints, the SuperTAP event system provides additional flexibility both in what can cause the emulator to intervene in code execution and in what actions can occur.

Feature overview

The emulator provides a powerful state machine that monitors the processor bus and the emulator's own counters, groups, and states. The system can track deeply nested sets of conditions, including recursive and reentrant code sections. These features provide powerful debugging capabilities for software debugging and for hardware/software integration.

- Up to 32 when/then statements can be defined at any time.
- Four event groups and four states provide the logical structure necessary for tracking deeply nested bugs.
- The event system includes four counters that can be monitored and controlled. Two of the counters are the processor's on-chip counters.
- Trace collection can be selectively controlled.
- Memory and register values can be monitored and modified.
- Emulation can be stopped before or after instruction execution.
- The event system can respond to or produce an external trigger signal, which can be used to trigger external devices such as logic analyzers or daisy-chained emulators.

Event system structure

The event system provides access and execution break control. It allows you to monitor a predefined set of conditions, called *events*, and then perform emulator *actions* based on those conditions by forming "when/then" statements.

The event system monitors emulator functions and target information at the bus cycle level, including every read or write cycle that the microprocessor executes, until the defined condition is encountered. When a defined condition in any active when/then statement is encountered, the emulator takes the specified action.

Up to 32 when/then statements are supported. When/then statements are active in all four event groups unless tied to a specific group when the statement is defined. Commands are provided to manage which group and which statements are active at any given time. Additional commands are provided to manage features used by the event system, such as the counters, and trace capture system.

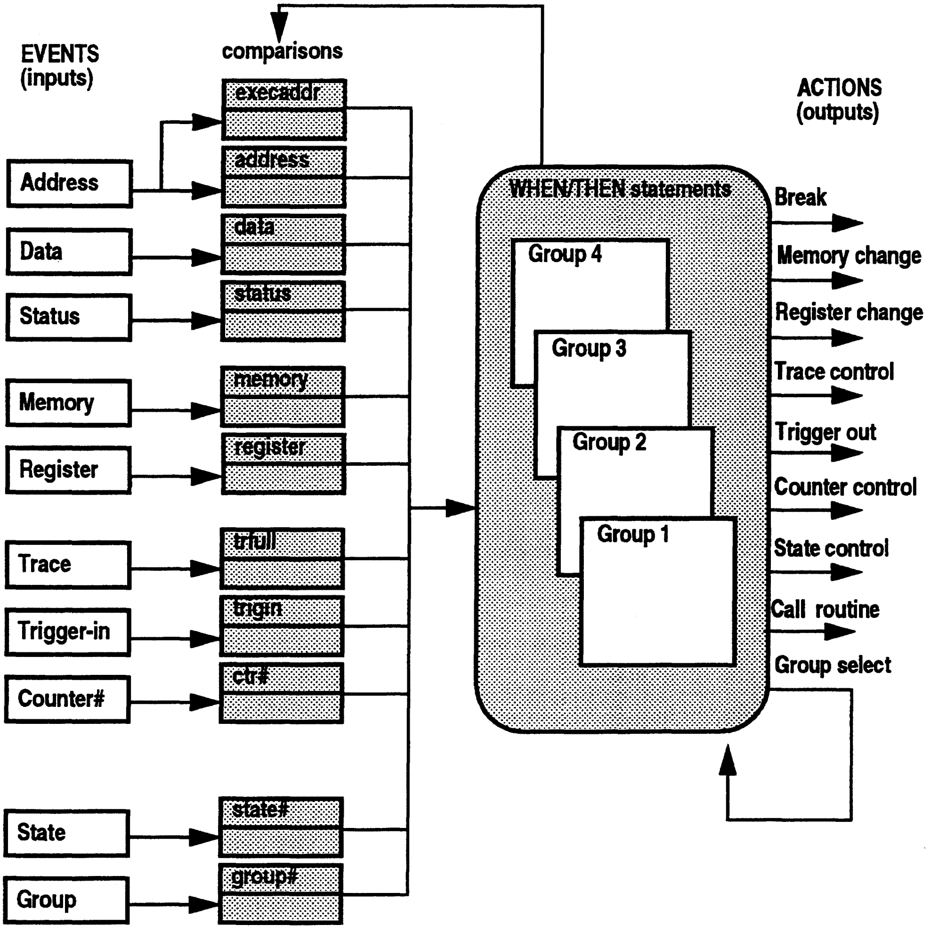


Figure 7-1 Event system structure

Event system commands

The following commands are used to define, control, or display event system statements.

To	Command	Abbreviation
Define when/then statement.	when	wh
Disable when/then statement.	whendisable	whend
Enable when/then statement.	whenenable	whene
Display or save event system setup.	whenlist	whenl
Clear when/then statement or setup.	whenclr	whenc
Display/specify event group active at run.	group	
Attach a named macro to a when/then break action statement.	breakcomplex	bc
Set/clear event system state flags.	state	

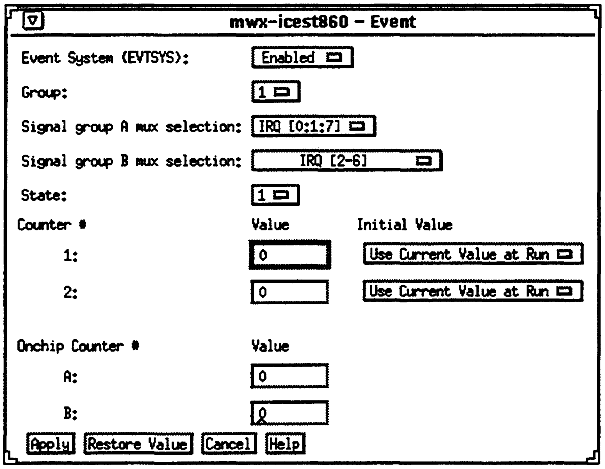
The commands above are documented in Help.

Setting event system options

You can configure event system options using the either the Command window or the Event configuration dialog box. These options are used to control or display the state of event system counters, display or set the current active group or state, or select the multiplexed signals that can be used as inputs to the trace and event systems.

➤ To open the Event dialog box

1. From the Windows menu, choose Emulator Configuration.
2. In the Emulator Configuration window, click Event.



mwx-icest860 - Event

Event System (EVTSYS): Enabled

Group: 1

Signal group A mux selection: IRQ [0:1:7]

Signal group B mux selection: IRQ [2-6]

State: 1

Counter #	Value	Initial Value
1:	0	Use Current Value at Run
2:	0	Use Current Value at Run

Onchip Counter #	Value
A:	0
B:	0

Figure 7-2 The Event dialog box.

➤ To set event options

- In the Event dialog box, select items from the option menus or enter values, then click the Apply button.

  You can leave the Event dialog box open while you create your event system statements.

  —or—

- From the Command window, type the name of the command and press return.

The following table lists the event system options.

To	Command
Display counter *n* value.	ctr*n*
Set the initial value of counter *n*. Only applies to the emulator counters (1, 2).	ctr*n*ival
Display all event state variables.	evtvars
Selects which multiplexed signals to use as inputs to the event system (irq[0, 1, 7] or lsa[0, 1, 7]).	siga_mux
Select which multiplexed signals to use as inputs to the event system (irq[2:6], lsa[2:6], wp, pcmcia, dp, rsv).	sigb_mux
Display/specify event group active at run.	group
Set/clear event system state.	state

Event system statements

The basic event system "when/then" statement is of the form:

when *event_expression(s)* **then** *action(s)*

When the *event_expression* occurs, the specified *actions* are taken. Event expressions and actions may include several logically related events.

Setting up when/then statements

The following sections provide an overview of event and action options. See the Help command descriptions for **when**, **when** *events*, and **then** *actions* for detailed information on when/then statement syntax and definition.

When *event_expression(s)*

An event expression is a logical combination of one or more events. Events can be combined or negated in the event expression. The following events can be used in an event expression.

To test when:	Use:
An address of the specified value appears on the address bus.	address
The specified counter value is reached in the numbered counter. Only applies to the emulator counters (1, 2).	ctr*n*
The specified data value appears on the data bus.	data
The instruction at the specified address is about to be executed.	execaddr
The specified group is active.	group*n*
The data at the memory address is as specified and another event condition has been met.	memory
The specified value appears in the register and another event condition has been met.	register
The specified status condition(s) appear on the bus.	status
The 32K trace buffer is full.	trfull

To test when:	Use:
A TTL trigger input of at least one bus cycle duration is received via the BNC trigger input connector on the back of the emulator.	trigin

Then *action(s)*

The emulator performs actions when the defined events are encountered while running the target system program. Multiple actions may be listed, separated by commas (,). The following actions can be used in the when/then statement.

When the event condition is true, to:	Use:
Stop program execution either just following an access or just prior to instruction execution. You can also specify whether you want to use on-chip, emulator hardware, or software breakpoints.	break ocbrk hwbrk swbrk
Zero, set to a specific value, increment or decrement the specified counter. Note: You can only set, increment, or reset the emulator counters (1, 2). The on-chip counters (A, B) can only be decremented.	ctr*n*rst ctr*n*set ctr*n*inc ctradec ctrbdec
Change to the specified group.	group*n*
Zero, set to a specific value, or increment the specified memory location.	memrst memset meminc
Zero, set to a specific value, or increment the specified register.	regrst regset reginc
Generate a TTL-level signal on the emulator trigger-out BNC connector.	trigout
Call and execute the specified target routine	call

When the event condition is true, to:	Use:
Turn trace capture on or off, or capture a single cycle.	tron/ troff/ trone

Examples of when/then statements

Below are examples demonstrating event system when/then statements. These examples cover commonly used event/action combinations. Refer to the two tables above for other events and actions.

- ❑ Break emulation when address 0x100 appears on the bus.

```
when address==0x100 then break
```

- ❑ Trace only the write cycles to the memory mapped port, led_port[0].

```
when ad==(&led_port[0]) && status==wr then trone
```

- ❑ Trace only the write cycles to the memory mapped port, led_port[0], when written to by a subroutine (outdot in the example).

```
when execaddr==outdot then group2
when execaddr==outdot_end then group1
when group2 && ad==(&led_port[0]) && status==wr then
trone
```

Valid event/action combinations

Most actions and events can be used alone or in combination. The exceptions are shown below:

Invalid combinations	Rules
when address && execaddr	Do not combine **address** and **execaddr** in the same statement using the AND (&&) operator. You can OR (||) the two events.
when *event* && execaddr then trone when *event* && memory then trone when *event* && register then trone	With **execaddr**, **memory**, or **register** events, do not use **trone**, **tron**, **troff**, **group*n***, **state*n***, **ctr*n*inc**. **ctr*n*dec**, or **trigout** actions.
when ctr*n* && ctr*n* then ...	A statement can include only one counter event.
when execaddr && status when execaddr && data	Do not combine the **execaddr** event with **status** or **data** events.
when execaddr==*start*..*end*	Do not use an address range with the **execaddr** event unless you are using on-chip breakpoints (**bptype onchip**).
when group*n* then ... when memory then ... when register then ...	The **group**, **memory**, and **register** events must be combined with another event using the AND operator (&&). Valid events are **address/execaddr/data/status/state*n*/trigin/trfull**.
when group*n* || group*n*	Multiple group events cannot be combined in the same statement.
when status *mnemonic* && *mnemonic*	Use a vertical bar (|) as the AND operator when you combine status mnemonics (do not use **&&**): when status *mnemonic* | *mnemonic*...

Qualifying event system memory accesses

All address comparisons must specify the transfer size and addressing mode appropriate for your target.

Address-based comparisons or actions can be qualified using the **address** and **size** commands with the **event** memory access type.

`ADDRESS EVENT` *mode* Determines whether an address is interpreted as logical or physical.

Note that this does not apply to on-chip events, which always use logical addresses. On-chip events are statements that use **ctr*x*dec** or **ocbrk** actions.

The **execaddr** and **memory** events and **memory** actions can override the **address** command setting within their syntax.

To override the **address** variable setting for the **address** event, use the AND operator (&&) to combine it with the **status** event.

The following table shows which qualifiers must be set for each event/action.

Event/ Action	In when/then statements, use:	In command settings, use:
`address`		`address event physical`
`execaddr`	`mode:`	`address event` *mode*

`memory`	`mode:`
	`data_grain`
		`address event` *mode*

where the memory qualifiers have one of these values:

`mode`	`logical_code`
	`logical_data`
	`physical`
		`logical_code`
		`logical_data`
		`physical`

Event/ Action	In when/then statements, use:	In command settings, use:
data_grain	/1	/1
	/2	/2
	/4	/4

Example

Change the value at logical address (**logical_code:**) 0xffe0 to FF (byte = **/1**) after counter 1 is 10:

```
when ctr1==10 then memset logical_code 0xffe0=0xff/1
```

This could also be done the following way:

```
address event logical_code
size event 1
when ctr1==10 then memset 0xffe0=0xff
```

Using the event system

Displaying and saving when/then statements

The current when/then statements can be displayed or saved using the **whenlist** command. To display the statements, use **whenlist** with no argument. A numbered list of statements is displayed.

To save the current when/then statements in a file use **whenlist** with the desired filename as an argument. For example, to save the current when/then statements to a file named ***my_events.inc***, enter:

```
whenlist my_events.inc
```

To use the when/then statements you saved with **whenlist**, choose Include Commands from the File menu, and select the name of the command file. You could also enter the **include** command in the Command window:

```
inc myevents.inc
```

If you want to include a command file each time you start the debugger, use the Startup Options Editor.

Enabling and disabling when/then statements

When/then statements are automatically enabled when they are created. They may be temporarily disabled and re-enabled.

Enter:	To:
`whendisable 3`	Disable statement 3.
`whendisable 2..4`	Disable statements 2, 3, and 4.
`whendisable 1,3,4`	Disable statements 1, 3, and 4.
`whendisable all`	Disables all when/then statements.
`whenenable 3`	Enable statement 3.
`whenenable 2..4`	Enable statements 2, 3, and 4.
`whenenable 1,3,4`	Enable statements 1, 3, and 4.
`whenenable all`	Enables all when/then statements.

Clearing when/then statements

When/then statements can be cleared using the **whenclr** command. When an event is cleared, it is removed from MWX-ICE.

Enter:	To:
`whenclr 3`	Clear statement 3.
`whenclr 2..4`	Clear statements 2, 3, and 4.
`whenclr 1,3,4`	Clear statements 1, 3, and 4.
`whenclr all`	Clear all statements.

Groups

A group is an exclusive state within the event system defined by **when group*n*** events and enabled by **then group*n*** actions. When a group is current, only associated when/then statements (and any global statements) can cause event system actions.

Changing groups activates the alternate set of events. Use the **group** command during pause to determine which group is currently active or to change the active group. At each **go,** the group specified by **group** becomes the current group. Other event groups become current dynamically by switching groups as a **group*n*** action of the event system.

Figure 7-3 illustrates the relationship between statements and groups. Statement #6 belongs to all four groups because no group was included in the statement definition.

```
#1 when address==end_init && group1 then group2
#2 when group2 && address==conveyer2 then group3
#3 when group3 && address==checkbelts && data==0x0004
   &=0x4 && status==rd then group4
#4 when group4 && address==beltjam..+0x400 &&
   status==rd then ctr1inc
#5 when group4 && ctr1==100 then break
#6 when trigin then break
```

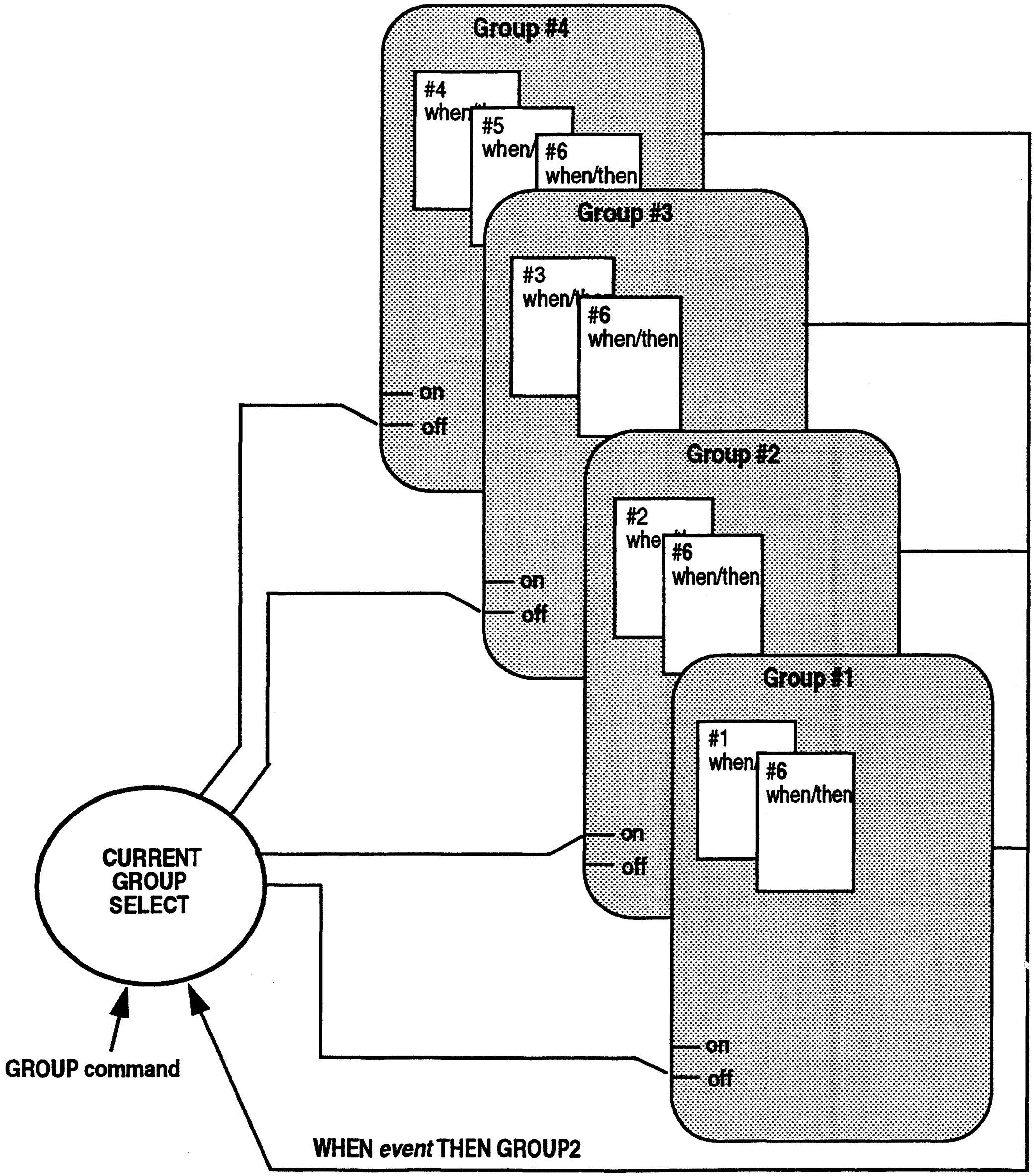


Figure 7-3 Event system groups

Example

As an example of the common use of two groups, you may wish to trace a subroutine after it has been called by module A or module B, but not if it has been called from modules C, D, or E. In this case, you would define a set of when/then statements to the address ranges of modules A and B. When either of these modules is encountered, switch to **group2** and look for the subroutine. After tracing the subroutine, switch back to **group1**.

```
trsys on
trace off
trinit current
when address==A..+100 || address==B..+80 && group1
then group2
when address==subroutine && group2 then tron
when address!=subroutine && group2 then group1,troff
```

General characteristics

- Statements are global (active in all groups) unless tied to a specific group in the **when** clause. Notice in Figure 7-3 that statement #6 appears in all groups.
- Among the four groups, no more than 32 when/then statements can be defined at one time.
- A group event must be paired with another event to be valid:

  ```
  when ad==0x100..0x1ff && group3 then tron
  ```

  —not—

  ```
  when group3 then tron
  ```

- Once a group change occurs, the system remains in that group until explicitly changed to a new group, even if the condition causing the group change is no longer true.
- Multiple group events cannot be combined in the same statement; for example,

  ```
  when group1 || group4
  ```

  is invalid.

- If you change to a group in which states are also defined, but do not specify a state as part of the group change action, the system defaults to state 1.

States

States can be used globally or within groups to create a second level of comparison. Like groups, states must be associated with other events to limit application of the event conditions to those times when the state is active. Unlike groups, states are only present as an active system resource when defined.

Care must be taken to prevent ambiguous states. This occurs most frequently when two events can appear on the bus simultaneously. Consider this example:

```
when addr==0x1000 then state1
when status==irq0 then state2
```

If the status comparison and the address comparison ever occur on the bus simultaneously, the state action is ambiguous. In such a case, the state actually applied is the one specified in the latter event statement.

In addition, if you change to a group in which states are also defined, but do not specify a state as part of the group change action, the system defaults to state 1.

Counters

There are two types of event system counters: emulator, and on-chip.

Emulator event counters

The 32-bit emulator event counters, counter 1 counter 2, are used to detect when events have occurred a certain number of times. The **ctr*n*** event tests for a counter value. The counters can be set (**ctr*n*set**), incremented (**ctr*n*inc**), and reset (**ctr*n*rst**) by when/then statement actions.

On-chip counters

The on-chip counters, counter A and counter B, cannot be used to directly define counter events. Instead, they are defined in event actions, where they count-down from a preset 16-bit value (**ctradec, ctrbdec**). To use these counters you must first load the counter with a value. When you use the on-chip counter action in an event statement, execution breaks when the counter equals zero. If you do not load the counter with a starting value, the counter counts down from zero, rolling over to 0xffff. From there it counts down to zero.

You can only use the on-chip counter action with an address event, an execaddress event, or an address and data event.

Examples

- In the following when/then statement, counter 1 is incremented whenever address is 0x1000:

  ```
  when ad==0x1000 then ctr1inc
  ```

- In the following when/then statement, emulation stops when counter 1 reaches 0x10:

  ```
  when ctr1==0x10 then break
  ```

- In the following commands, emulation stops when counter A is zero.

  ```
  ctra 10
  when ad==0xfc then ctradec
  ```

Related commands

For counter 1 and 2, the **ctr*n*ival** command sets the initial state of a counter from the command line. At pause to run transitions the counter can be reset to zero or it can retain its current value. The **ctr*n*** command displays the current counter value. You can use the **ctr*n*** command to load the on-chip counters A and B.

General characteristics

- A when/then statement cannot contain two different counter actions. Use separate statements for the counter events.
- If you use the on-chip counter action with an **execaddress** event, then execution breaks when the counter value is one.

Memory and registers

The event system can monitor and change memory and register values.

Memory and **register** events are always used in combination with other events. Valid events are **address/data/status/ state*n*/trigin/trfull**. This allows you to check for the value of a memory location or variable, or a register after another set of conditions has been met.

Memory and register values can be changed as an event system action.

To:	Use:
set a memory value to zero	memrst
increment a memory value	meminc
set a memory value to a new value	memset
set a register value to zero	regrst
increment a register value	reginc
set a register value to a new value	regset

Each read or write to a memory location or a register requires a run-pause-run transition. For this reason, associating these events or actions with frequent events affects real-time operation. See page 7-23 for an discussion of real-time operation.

External triggers

Trigger input

Event system trigger events are true when a TTL-level trigger input of at least one bus cycle duration is received via the BNC trigger input connector on the back of the emulator.

Trigger output

The event system provides one TTL-level trigger output to generate a pulse at an external BNC connector when an event becomes true. The BNC connection may be used to trigger another emulator, an external scope or a logic analyzer on an emulator event. Normally, the voltage level on this connector will be high (+5V); an active trigger will drive the voltage low for a minimum of one CPU bus cycle.

Examples

- To program the event system to output a trigger signal when address 0x34 is on the bus, enter the following:

  ```
  when address==0x34 then trigout
  ```

- To program the event system to trace once cycle when a trigger input is detected, enter the following:

  ```
  when trigin then trone
  ```

Attaching macros to event system breakpoints

The **breakcomplex** (**bc**) command associates a macro with an when/then statement that uses the **break** action, creating a 'complex' breakpoint.

Examples

Determine the when/then statement number by entering **whenlist** to display the numbered list of when/then statements:

```
1. when ad==0x0..3ff then group2
2. when ad==0x9000 && status=wr then break
3. when ad==0x9000 && data==0xfffe then break
```

- To invoke my_macro after the **break** action in when/then statement #2, enter:

  ```
  breakcomplex 2 ;my_macro()
  ```

- To execute a trace-dumping macro, dumptrc(), after the event system access break on the address plus data value in statement #3, enter:

  ```
  breakcomplex 3 ;dumptrc()
  ```

Complex breakpoint process

With a complex breakpoint, each time the specified event system break is encountered, the debugger:

1. Completes the execution of the instruction at that location.
2. Suspends program execution.
3. Executes the macro. Depending on the macro, the debugger will do one of the following:

   If the macro return value is true (non-zero), the debugger resumes execution at the instruction immediately after the break.

   If the macro return value is false (zero), the debugger returns to command mode and displays break information.

Event system operation

Break action latency

Several factors determine the number of cycles beyond a break action that the processor may run before it stops. Processor speed, the complexity of the instruction being executed at the time a breakpoint is detected, the DPI clock frequency, and the type of memory being used can all affect how great this latency may be.

For a **break** action caused by an **execaddr** event, the break occurs before the instruction is executed.

For a **break** action caused by all other events, the break is initiated in the next bus cycle after the event occurs. Several additional bus cycles may occur and show in raw trace as a few additional frames of trace beyond the break condition.

The break signal appears in trace memory where the break is initiated. The bus cycles occurring prior to completion of the **break** action follow the break request, and are marked with the letter B in raw trace.

On-chip access breaks (**ocbrk**) occur immediately *after* the load/store instruction has been executed.

Real-time operation

Many event system statements are processed in real time. For certain event and action combinations, the emulator requires a run-pause-run transition to test the event or perform the action specified by a when/then statement. During this time, real-time execution is affected.

The **rte** command controls whether real-time operation is enforced. When **rte** is on, when/then statements that impact real-time execution cannot be defined. If any non-real-time events are enabled, **rte** cannot be set to on. Disable or clear any non-real-time events before turning **rte** on. During dynamic run mode (**drun**), when/then statements may be entered and processed.

The following table shows which event/action combinations result in real-time or non-real-time operation. Not all combinations are valid; see "Valid event/action combinations" on page 7-11 for exceptions.

these events	**combined with these actions**	**result in**

`address`
`data`
`status`
`group`*`n`*
`state`*`n`*
`trfull`
`trigin`
	`trigout`
	`trone/tron/troff`
	`break/ocbrk/hwbrk/swbrk`
	`group`*`n`*
	`ctr`*`n`*`rst/ctr`*`n`*`set/ctr`*`n`*`inc`
		real-time operation

`execaddr`	`break`	**real-time operation**

`memory`
`register`
`ctr`*`n`*
	`trigout`
	`trone/tron/troff`
	`break/ocbrk/hwbrk`
	`ctradec/ctrbdec`
		non-real-time operation

`all`	`memrst/memset/meminc`
	`regrst/regset/reginc`
	`call`
		non-real-time operation

`execaddr`	`all actions EXCEPT`
	`break`
		non-real-time operation

Memory and register events, which must be paired with another event, are not evaluated until the other event is true. To extend real-time operation, avoid pairing memory and register events with events that are always true or with other non-real-time events.

Address events for code accesses

When using an **address** event for a code address that is in a cached region, the address will appear on the bus during a prefetch. When this occurs, the specified actions will begin. This occurs prior to execution of the code. Actions will occur even if the code at **address** is never executed.

On-chip caches

When the processor on-chip instruction and data caches are used, you must make sure that instruction or data show cycles enabled, or the instruction/data caches will cause some *access* (address and data) events to be missed when cache hits occur. If copy back caching is enabled for the data cache, access events may trigger in sections of code having nothing to do with the access; it is just where the processor decided to copy back the cached data to memory. In MWX-ICE, you can use the **showinst** command to enable instruction show cycles (**showinst indirect**). The data show cycles are enabled by setting the DSHW bit in the processor's SIUMCR register.

Event system resources

The event system uses both hardware, software, and on-chip resources to test for and act on the events you define. The same emulator resources are used to support both breakpoints and the event system.

MWX-ICE allows up to 32 when/then statements to be defined. Depending on the events and actions specified, different numbers of emulator resources are required by each when/then statement.

In general the emulator manages these resources, warns you when it makes adjustments, and presents an error when resources are exhausted or when you attempt something that creates a conflict. Only if you are frequently exhausting resources do you need to review the information in this section.

Resource allocation

Definitions

The following terms are used to explain **hardware** resource allocation. The term *ads.event* stands for an address, data, status event.

ads.event

Any event expression that uses the AND operator (&&) to combine address, data, or status events. Event expressions that are combined using the OR operator (||) are considered to be separate *ads.events*. For example, four *ads.events* are shown below:

```
when ads.event1 || ads.event2
when ads.event3
when ads.event4
```

simple ads.event

An *ads.event* with no address/data ranges or negations. For example:

```
when ad==0x0034 &=0x00ff
when status==rd
when ad==0xfe00 && status==rd
when ad==0xfe00 && data==0x5555 && status==rd
```

complex ads.event

An *ads.event* with address ranges or negations. For example:

```
when ad!=0xfe00..0xfeff && status==rd
```

hardware event comparator

Hardware comparators in the emulator used for detecting when/then statement **address**, **data**, and **status** events and access breakpoints.

Resource allocation

- The system makes available 16 hardware event comparators.
- *Simple ads.events* use one event comparator.
- *Complex ads.events* that include only ranges, use two event comparators.
- *Complex ads.events* that include address/data negations with no mask and a status event, use four event comparators.
- *Complex ads.events* that include address/data negations with a mask and a status event, use multiple event comparators, depending on the mask value.
- Only enabled when/then statements, whether global (no group in the when clause) or associated with a single group, use hardware event comparators.
- Access breakpoints (**ba**, **br, bw**) consume an event comparator, limiting the comparators available for event system use.
- A total of 50 breakpoints or events can be defined.
- Up to 10 access breakpoints can be defined.
- Up to 32 event statements can be defined.
- Up to 16 actions can be included in a single when/then statement.

Using resources wisely

- When/then statements are automatically enabled when defined. Use **whendisable** to disable any when/then statement that does not apply to the current portion of the debugging session. Use **whenenable** to re-enable when/then statements.
- If event resources run short, save (**whenlist *filename***) and clear (**whenclr**) any when/then statement that does not apply to the current portion of the debugging session. This frees its allocated resources for other events.

Additional information

- The tutorial in Chapter 10 offers additional practical examples.
- Help provides detailed descriptions of all the commands mentioned in this chapter, as well as structured "browse sequences" that organize the Help topics by subject.

Chapter 8
Support for MPC8XX Registers

The SuperTAP system enables display and modification of all MPC8XX family registers.

Using the set of initialization registers

The SuperTAP maintains a copy of the values stored in some of the processor's chip-select and pin assignment registers. This copy is called the set of *initialization registers*, or INITREGS, because it is used whenever you resume operation after a target-generated or MWX-ICE command-driven processor reset. The INITREGS must match the values that the processor chip-select and pin-assignment registers have after you run your initialization code.

Why do I need to set up the INITREGS?

The INITREGS feature provides a way of decoding the multiplexed control pins, disabling the software watchdog timer, and making MWX-ICE memory operations possible immediately after a processor reset. For these reasons, you must specify the INITREGS at the beginning of each session as described in the following sections.

Because the processor multiplexes several control signals on the same pin, it is difficult to determine what functions the pins have been programmed for. The processor's chip-select and pin-assignment registers provide the information needed to decode the control pins and address lines, but they are unavailable while the processor is running. In addition, these registers are cleared whenever the processor is reset.

Note that, when discussing the initial registers, there are *two* items which are being saved: the list of registers to manage, and the list of the *values* of those registers. In most contexts below, the discussion will focus on manipulating the list of values, except where noted. Unless you specify a different list of registers, the system-default list will be used.

How do I turn the INITREGS feature on or off?

In the command window, type the command **initregs on** to enable this feature (the default is **on**). Once you have set up the INITREGS for your target, you should leave this feature on except when you are debugging the boot code itself. The command **initregs off** disables this feature.

How do I see the current "state" of the INITREGS feature?

Enter the command **initregs** (without an argument) to show the current state of this feature.

Where are the INITREGS values stored?

When you use the **initregs save** command, MWX-ICE reads the processor registers and creates an internal copy of the registers and then saves the register values to the file specified, or to the default file, if no file was specified. You can edit this file to change values, or to add or remove registers from the list. You should use care when editing this file, however, as some of the registers can only be written once, and the order the registers appear in the file can be important. For example, the UPM*x* registers should be initialized before the M*x*MR is written to turn on DRAM refresh, which immediately uses UPM*x* values.

When you save or restore the register values without specifying a path and filename, MWX-ICE looks in specific directories for a data file. The name of the file is **iregs*xxx*.dat** (where the *xxx* is replaced by your processor type: 860, 821, and so forth). When the INITREGS feature is enabled (**on**), MWX-ICE automatically uses the registers and values from this file.

MWX-ICE searches the directories for the INITREGS data file in the following order:

- *current_directory*\amc\ST8XX\**iregs*xxx*.dat**
- $XRAYMASTER\amc\ST8XX\**iregs*xxx*.dat**
- C:\ST8XX\amc\ST8XX\**iregs*xxx*.dat**

How do I set up the initregs?

There are two ways to set up the INITREGS:

- ❑ Run boot code that sets up the chip-select and pin-assignment registers, and then copy the processor's registers into the INITREGS.
- ❑ Edit a copy of the **iregs*xxx*.dat** file to have the correct values for your target. You can then restore the INITREGS from this file.

Running boot code in ROM

If you have startup code in your target ROM, you can run this code to set up the chip-select and pin-assignment registers. Once you've run your initialization code, you can then copy the processor's registers into the INITREGS.

(Note that the following commands work with the system-default list of registers).

➤ **To set up the initial registers:**

1. Set up your environment and start MWX-ICE as described in Chapter 2. Be sure to read the section "Software watchdog timer" on page 2-23.
2. Run your initialization code: type **go** in the Command window or click the Go button.
3. Click the Stop button.
4. Type **initregs on** (unless this has already been done).
5. Type **initregs save** [*filename*] to make a permanent copy of the initregs in a file.

The file saved is an initregs data file. It can be edited to change either the list of registers or their values. If you don't specify a filename, the name **iregs*xxx*.dat** is used (where the *xxx* is replaced by your processor type: 860, 821, and so forth). This file is created in one of the directories listed in "Where are the initregs values stored?" on page 8-3.

Editing a copy of the iregs*xxx*.dat file

If you don't have boot code in ROM to run, you can edit the initregs file to match your target system. Making a copy of the default initregs data file is a versatile way to manage your chip select and pin assignment registers if you need to change them frequently. Using several initregs data files, you can change the values for these registers without having to type in a complete register set each time you want to change them.

Note

The default file is called **iregs*xxx*.dat.def** (where the *xxx* is replaced by your processor type: 860, 821, and so forth). The default file is located in *install_dir*\amc\ST8XX. The original **.def** file should never be modified. This file sets up the registers for *isolation* mode. In isolation mode the SuperTAP is not connected to a target. In the same directory, there are three other initregs files: **iregs860.dat.ads** and **iregs860.dat.amc.** These files set up the registers for use with the Motorola ADS board and the Applied Microsystems test target. To use those files, just make a copy of them with the name **iregs860.dat.** The third file is **iregs860.dat.all**; it lists almost every configuration register for the MPC860. To use this file, read the comments in the file and edit it as necessary. Then make a copy with the name **iregs860.dat**.

Use the following steps to edit the **.dat** files.

➤ **To edit the INITREGS data file:**

1. Copy the default **iregs*xxx*.dat.def** file and save it as **iregs*xxx*.dat** in the *install_dir*\amc\ST8XX directory.
2. Edit the **iregs*xxx*.dat** file as plain (ASCII) text with any text editor or word processor.
3. Define each register on a separate line.

Choose the registers and values appropriate for your system. To add comments, begin a line with the pound sign (#).

```
# MPC860 Initregs file
# Isolation mode setup.
# Customize for a specific target
#
# Turn off the MPC860 internal
# software watchdog timer
SYPCR=0xffffff80
# Set the Recoverable Interrupt
# so that maskable breaks work
MSR=0x42
    .
    .
    .
```

4. Save the file as plain ASCII text.

   With the initial-register feature enabled (**initregs on**), the file **iregs*xxx*.dat** is automatically loaded following a reset. You can also save the file using a different filename. If you use a different filename, you need to reload the register values explicitly using the **initregs restore** command.

Restoring the initial-register values

If you've set up and saved the INITREGS for your target using the default filename for your processor (**iregs*xxx*.dat**), you won't need to explicitly restore the values from the file as long you've set **initregs on** and saved your configuration using **consave** or the Emulator Configuration window.

If you've used a filename or path that is different from the default you'll need to restore the values from the file using the **initregs restore** command. Once restored, those registers and values are used following a processor reset.

➤ **To restore the initregs:**

1. If you haven't already done so, enable the initregs feature. From the Command window, type **initregs on**
2. Type **initregs restore** *filename*
3. Reset the processor, type **reset** in the Command window.

Note

Be sure to use **initregs restore** command instead of the **include** command to restore INITREGS.

Using the initregs command

The **initregs** command controls the updating of the system interface unit (SIU) registers, and others, from the INITREGS after an MWX-ICE **reset** command or a target-generated processor reset. Once you have initialized your INITREGS, leave the feature enabled (**initregs on**) unless you are debugging the boot code itself.

Syntax

```
initregs [option]
```

The options to the command are as follows:

Option	Description
off	After a processor reset, the registers come up in their default reset state. No attempt is made to reprogram them.
on	Loads the INITREGS values into the processor upon a processor reset. This is the default.
<none>	Shows whether or not the feature is enabled.
save	Writes the INITREGS to a file. Does not affect the processor's register values.
restore	Reads INITREGS from a file. Does not affect the processor's register values. This option is the *only* way to change the list of registers from the default list.

Enabling the INITREGS feature

You can enable the initregs feature from the command line while MWX-ICE is running.

➤ **To enable INITREGS**

1. Set up and initialize the initial registers as described in previous sections.
2. From the Command window, type **initregs on**
3. If you want to program the processor registers with the INITREGS values, reset the processor by typing **reset** in the Command window.

What happens when the INITREGS feature is enabled?
At power up and after an MWX-ICE or target-generated processor reset, the chip-select information in the processor's register set is reprogrammed.

With **initregs on**, the reset sequence is as follows:

1. Reset the processor.
2. Copy all initial registers, if initialized, to the corresponding processor registers, which enables chip-selects, etc.

Disabling the INITREGS feature

You can disable the INITREGS feature from the command line while MWX-ICE is running. To start MWX-ICE with INITREGS disabled, save the configuration to startup.inc.

➤ **To disable INITREGS:**

- From the Command window, type **initregs off**

What happens when the INITREGS feature is disabled?
The command **initregs off** allows the target software to configure the chip selects rather than having them pre-configured at reset. This is useful when debugging boot code.

Support for MPC8XX family registers

MWX-ICE provides direct, point-and-click access to all registers of the supported processors.

Setting up the register sets

Before emulating, you must configure the emulator and debugger for the processor to be used. This also controls which registers can be accessed.

To begin emulation, you must select the correct processor at MWX-ICE startup using the MWX-ICE Startup Options Editor. The Startup Editor saves your selections to a file (MWX.CFG is the default file). The debugger looks for this file at startup.

➤ **To configure the processor**

1. Click the Start Menu, point to Applied Microsystems, then point to MWX-ICE SuperTAP, then click, the MWX-ICE Startup Options Editor.
2. Select the type of processor being emulated from the Processor list.
3. Specify additional startup options, if needed.
4. Choose OK.

Viewing and modifying MPC8XX family registers

Once a MWX-ICE is configured for the correct processor, you can use a variety of tools to access the registers.

Viewing registers

There are three methods to view registers:

Search Keywords:
Register window
Register mnemonics
SETREG

- From the Displays menu, use the Registers menu to bring up a window displaying the group of registers of interest. Depending on the processor, you can select among general, memory control, system control, CPM, MMU, PCMIA, UPM, and other groups.
- Use the CPU Browser to get error-protected, point-and-click display and control of all specialized register bits. From the Displays menu, select CPU Browser; then select the register(s) of interest.
- Use the **cexpression** command to display individual registers.

Modifying registers during pause

There are three methods for modifying registers:

- Click the register in the Registers window to bring up a dialog box that accepts new values.

  —or—

- Within the CPU Browser windows, use the bit-configuration menus to select options for the registers.

  —or—

- From the command line, use the **setreg** command to enter a new value:

  ```
  setreg @register_name=value
  ```

For detailed explanations, see Help for descriptions of the Registers and CPU Browser window and the **setreg** command.

Note

When the CPU Browser window is open, all entries made using any register controls are checked for errors. If the new values are entered using **setreg** or the registers windows, the value is applied before the error dialog is presented. If the change is made using the CPU Browser, the new value is not applied until you respond to the warning dialog. The exception is an invalid value that is corrected by the processor core itself. Invalid entries in some registers do not trigger an warning because the processor ignores the invalid bits. With **rgverify on**, a warning is generated when the value written is not applied to the register.

Modifying registers during operation

The event system provides means to test for register values and to set, increment, or clear registers as the result of specified conditions. Chapter 7 covers event system features and use.

Chapter 9
MWX-ICE Command Quick Reference

MWX-ICE is based upon the Microtec Research XRAY simulator (XHS). MWX-ICE supports most XRAY commands and adds commands to support emulation. This chapter lists the commands by function and indicates which of these commands are supported by both MWX-ICE and XRAY and which simulator commands are not valid in MWX-ICE.

Check the index if you are unsure of the category for a specific command. See the following section on Help for information on how access to complete command descriptions.

Help

Complete command descriptions are provided in the online help system. Command descriptions may be found several ways:

If you...	Do this...
Don't know the command name	Click Contents Click Keyboard Commands (Functional Groups) Locate appropriate functional group Click *command_name* —or— Click Search Search for topic, functional name —or— Click Contents Click on appropriate feature/function Look for commands mentioned within text
Know the command name	Click Contents Click Keyboard Commands (Alphabetical Listing) Click *command_name* —or— Click Search Enter *command_name* in the Search box Click Find Topics button Click *command_name* Click Go To button

See Chapter 2 for an overview and for information on starting Help without starting MWX-ICE.

Command groups

Each category begins with a brief description of the functions of that group. The full command name is bulleted at the left. The description and syntax column provides a brief explanation of the commands function and a full expression of all syntactical elements. If an abbreviation exists for the command, it is used in the syntax example. Because not all commands can be used with MWX-ICE or the XHS simulator, identifiers (■) are placed in the columns that apply.

Entering commands

From the Command window, you type MWX-ICE commands in the Enter Command box and then click the Enter Command button or press <Return>. Any combination of upper-case and lower-case letters can be used in commands. Some functions can also be performed in specialized windows or menus.

Session control

These commands provide session-level functions.

Command	Description/Syntax	MWX-ICE and XHS
journal	Records a debugger session in a file. `j [/a] [off	on="filename"]`	■
log	Records debugger commands and errors in a file. `log [/a] [off	on="filename"]`	■
mode	Selects debugger mode (high or assembly). `m [high	assembly]`	■

option	Sets debugger options for this session.
	`OP [option_name = {value|symbol}]`
	Options and arguments:
	`ALIGN={ON|OFF}`
	`ANIMATE={ON|OFF}`
	`CPU=Read-only display of processor`
	`DEMANDLOAD={ON|OFF}`
	`EMULATOR=Read-only display of boot option`
	`FRAMESTOP={ON|OFF}`
	`INCECHO={ON|OFF}`
	`LINES={ON|OFF}`
	`LOADDEFAULTS={LOAD command options string}`
	`RADIX={HEX|DECIMAL}`
	`SYMBOLS={ON|OFF}`
	`TYPECHECK={ON|OFF}`
		■

quit	Terminates a debugging session. `Q [y]`	■
setstatus dir	Sets the current directory. `ss dir [=new_local_dir]`	■
setstatus environment	Overrides the XRAY environment variable. `ss env[="path1:path2:path3"]`	■

Window control

These commands enable configuring, calling up, and customizing the debugger windows.

Command	Description/Syntax	MWX-ICE	XHS
mode	Selects debugger and display mode (high or assembly) for active window. `m [high	assembly]`	■	■
vactive	Activates a window. `va {window_number}`	■	■
vclear	Clears data from a window. `vc [window_number]`	■	■
vclose	Removes a user-defined window. `vclo [window_number]`	■	■
vmacro	Attaches a macro to a window. `vm window_number [,macro_name()]`	■	■
vopen	Creates a window or changes sizes. `vo [window_number, screen_number, tr, lc, br, rc]`	■	■
vsetc	Sets the cursor position for a window. `vse window_number, line, column`	■	■

Memory and file handling

These commands enable configuration, display and modification of target and emulator memory and uploading and downloading of files.

Command	Description/Syntax	MWX-ICE	XHS
Amd*device*	Specifies the flash memory to be programmed. (case-sensitive) Amd*device1*(*base_addr*,"*space*",*width*) Amd*device2*(*base_addr*,"*space*", *by8or16*,*width*)	■	
address	Specifies logical or physical addresses for memory accesses. ADDRESS *access_type* [logical_code|logical_data| physical]	■	
asm	Single line assembler. asm [org *address*] asm [*mnemonic* [*operand*][, *operand*]...]	■	
compare	Compares two blocks of memory. COM [/R] [*address_range*, *address*]	■	■
copy	Copies a memory block. COP *address_range*, *target_address*	■	■
crc	Calculates a CRC for a range of memory. CRC *address_range* [/*memory_space*]	■	
disassemble	Displays disassembled memory (assembly mode). disa [*address*]	■	■
dnl	Downloads non-DWARF/ELF hex file to target. dnl "*filename*" [,*offset*]	■	
dnlfmt	Specifies download format. dnlfmt *format*	■	

Command	Description/Syntax	MWX-ICE	XHS
dnl_gap	Specifies maximum bytes between contiguous blocks in downloads. dnl_gap [0-1024]	■	
dump	Displays memory contents. du [/b | /w | /l] [*address* | *address_range*]	■	■
EraseDevice	Erases the device identified by Amd*device* or Intel*device*. (case-sensitive) EraseDevice()	■	
error	Sets include file error handling. er = {quit | abort | continue}	■	■
expand	Displays all local variables of a procedure. exp [*stack_level*] [,*window_number*]	■	■
fill	Fills a memory block with values. fil [/b | /w | /l] *address_range* [= {*expression* | *expression_string*}]	■	■
fopen	Opens a file or device for writing. fo [/a] [/r] *window_number*, "*filename*"	■	■
fprintf	Prints formatted output to a window or file. f *window_number*, "*format_string*" [,*argument*]...	■	■
include	Reads in and processes a command file. inc "*filename*"	■	■
initregs	Initializes processor chip-select and pin assignment registers. INITREGS [ON | OFF] INITREGS [SAVE [*filename*]] INITREGS [RESTORE [*filename*]]	■	

Command	Description/Syntax	MWX-ICE	XHS
Intel*device*	Specifies the flash memory to be programmed. (case-sensitive) Intel*device1*(*base_addr*,"*space*", *width*) Intel*device2*(*base_addr*,"*space*", by8or16 ,*width*)	■	
list	Displays source code. l [*line_number* | *procedure_name* | @*stack_level*]	■	■
load	Loads an object module for debugging. loa [/a][/c][/ni][/np | /sp][/ns] *absolute_filename* [,*root*] [&*base_addr*] [;*section* [,*section*]...]	■	■
LockDevice	Disables flash programming (case-sensitive). LockDevice()	■	
map	Defines a region of overlay or target memory. MAP *address_range* | *start..+length* [,*mode*][=type]	■	
mapclr	Clears overlay memory map. MAPCLR	■	
maplist	Displays or saves current memory mappings. MAPLIST [*filename*]	■	
memvars	Displays memory access variable values. MEMVARS	■	
overlay	Selects target or overlay as memory for various actions. OVERLAY *access_type* [ON|OFF]	■	
ovreadthru	Controls whether reads to overlay also go to target memory. ovreadthru [ON|OFF]	■	
ovwritethru	Controls whether writes to overlay also go to target memory. ovwritethru [ON|OFF]	■	

Command	Description/Syntax	MWX-ICE	XHS
reload	Reloads absolute file image. rel [/c] [/d] [/r] [*root*]	■	■
RemoveDevice	Removes flash device specification. (case-sensitive) RemoveDevice()	■	
restart	Resets program counter to the program starting address. rest	■	■
restore	Restore memory (XHS) and registers (MWX-ICE and XHS) from a file. resto [*save_filename*]	■	■
rgverify	Enables verification of writes to registers. rgver [on | off]	■	■
save	Saves current memory (XHS) and registers (MWX-ICE and XHS) to file. sa [*save_filename*]	■	■
setmem	Changes the values of memory locations. sm [/b | /w | /l] *address* [={*expression*|*expression_string*} [,{*expression*|*expression_string*}] ...]]	■	■
setreg	Changes the contents of a register. sr *@register_name =value*	■	■
size	Sets the size for memory accesses. size [*memory_access_type* [1|2|4]]	■	
test	Examines memory area for invalid values. te [/b | /w | /l] [/r] [*address_range* [= {*expression* | *expression_string*}]]	■	■
UnlockDevice	Enables flash programming. (case-sensitive) UnlockDevice()	■	

Command	Description/Syntax	MWX-ICE	XHS
upl	Uploads non-DWARF/ELF hex file to host. upl "*filename*",*address_range*	■	
uplfmt	Specifies upload format: Intel, SREC, XTEK (extended Tektronics hex). uplfmt *format*	■	
verify	Memory read-after-write verify switch. verify [on | off]	■	
when	Breaks on memory or register values; alters memory or register values as an event system action. WHEN *event* && memory *address=value* then ... WHEN *event* && *register=value* then ... WHEN *event* then memrst|meminc|memset *address*[=*value*] WHEN *event* then regrst|reginc|regset *register_name*[=*value*]	■	
xlate	Converts logical address to physical address. XLATE *address*	■	

Controlling execution and using breakpoints

These commands provide basic execution control and access and instruction breakpoints for initial debugging.

Command	Description/Syntax	MWX-ICE	XHS
bptype	Specifies the type of breakpoint (software, on-chip, or emulator hardware). `bptype [hw|sw|onchip|choose]`	■	

breakaccess	Sets an access breakpoint.
	`BA [`*address* `|` *address_range*`]`
	`[;`*macro_name*`()]`
		■	■

breakcomplex	Attaches a macro to an event system break. `BC` *n* `;`*macro_name*`()`	■	■

breakinstruction	Sets an instruction breakpoint.
	`bi [`*address* `|` *address_range*`]`
	`[;`*macro_name*`()]`
		■	■

breakread	Sets a read access breakpoint.
	`BR [`*address* `|` *address_range*`]`
	`[;`*macro_name*`()]`
		■	■

breakwrite	Sets a write access breakpoint.
	`BW [`*address* `|` *address_range*`]`
	`[;`*macro_name*`()]`
		■	■

clear	Clears a breakpoint. `cl [`*breakpoint_number* `|` *breakpoint_number_range*`]`	■	■
drun	Starts executing and enter dynamic run mode. `DRUN`	■	
dstop	Exits dynamic run mode. `DSTOP`	■	
dupdate	Starts polling and sets frequency in dynamic run. `DUPDATE [`*n*`]`	■	

Command	Description/Syntax	MWX-ICE	XHS
go	Starts or continues program execution. g [=*start_address*[,]] [*temp_break* [%%*passcount*[,]]...] [;*macro_name*()]	■	■
gostep	Executes macro after each instruction step. gos *macro_name*()	■	■
pause	Pauses simulation or emulation for specified seconds. pa [*n*]	■	■
poreset	Resets the processor with a power-on reset. poreset	■	
reset	Resets PC and synchronizes with target reset. reset	■	■
restart	Resets the program starting address. restart	■	■
serial_core	Controls serialization of CPU Core. serial_core [ON|OFF]	■	
sit	Stops in target loop address. sit [*address*]	■	
sitstate	Selects emulation stop method. sitstate [ON|OFF]	■	
step	Executes a specified number of instructions or lines. s [=*start_addr* [,*value*] | *value*]	■	■
stepover	Steps, but executes through procedures. so [=*start_addr* [,*value*] | *value*]	■	■

Capturing and displaying trace

These commands capture bus conditions and display them in assembly, C-source or mixed source-assembly.

Command	Description/Syntax	MWX-ICE	XHS
drt	Displays raw trace. DRT [*start_line* | *start..end_line*]	■	
drtdata	Limits display of the data bus. DRTDATA [NOTALL|ALL]	■	
drtfull	Displays all fields in trace frames. DRTFULL [ON|OFF]	■	
dt	Displays disassembled trace. DT [*start*|*start..end*]	■	
dtb	Displays disassembled trace from latest to earlier. DTB	■	
dtf	Displays disassembled trace from earlier to later. DTF	■	
dxinsert	Interleaves raw and disassembled trace. DXINSERT[ON|OFF]	■	
dxlabels	Shows symbols for branch destinations. DXLABELS [ON|OFF]	■	
ppt	Traces peeks and pokes. PPT [ON|OFF]	■	
showinst	Provides show cycle control. SHOWINST [NONE|INDIRECT|FLOW|ALL]	■	
siga_mux	Selects IRQ (0, 1, 7) or External LSA Bits (0, 1, 7). SIGA_MUX [IRQ | LSA]	■	
sigb_mux	Selects IRQ (2:6), RVS*, DP(0:3), LSA bits (2:6), or WP. SIGB_MUX [LSA|WP|PCMCIA|DP|RSV|IRQ]	■	

Command	Description/Syntax	MWX-ICE	XHS
timclk	Sets the timestamp resolution. `TIMCLK [40ns|200ns|1us|10us|100us|1ms|10ms|100ms]`	■	
trace	Dynamically enables/disables trace capture. `TRACE [ON|OFF]`	■	
trbase	Specifies offset timestamp base frame. `TRBASE` *`frame_number`*	■	
trclr	Clears trace buffer. `TRCLR`	■	
trdisp	Sets trace display to assembly, source, or both. `TRDISP [ASM|SRC|BOTH]`	■	
trcext	Enables external trace cycles `TRCEXT [ON|OFF]`	■	
trframes	Displays number of trace frames in buffer. `TRFRAMES`	■	
trinit	Selects trace initial state at run. `TRINIT [ON|OFF|CURRENT]`	■	
trqual	Selects bus or clock cycle capture. `TRQUAL [BUS|CLK|DXQUAL]`	■	
trrunclr	Enables/disables trace clear-on-run. `TRRUNCLR [ON|OFF]`	■	
trstamp	Controls timestamp display in trace. `TRSTAMP [INTERVAL|OFFSET]`	■	
trsys	Provides dynamic control of trace subsystem. `TRSYS [ON|OFF]`	■	

tsrch	Search trace memory for patterns.
	`TSRCH [`*`trace_range`*`],`
	`[addr=`*`value`*`[&=`*`mask`*`]]`
	`[&& data=`*`value`*`[&=`*`mask`*`]] [&&`
	`stat=`*`mnemonic`*`]`
		■	

Programming the conditional event system

These commands provide the setup and control functions for the conditional event system provided by the SuperTAP. With these you can specify nested sequences of possible target conditions and then trigger various responses.

Command	Description/Syntax	MWX-ICE	XHS
breakcomplex	Executes a macro following an event system break. BC *n ;macro_name*()	■	
ctr*n*	Displays counter n value. CTR*n*	■	
ctr*n*ival	Specifies initial value of counter n at run. CTR*n*IVAL [CURRENT|RELOAD]	■	
evtvars	Displays internal debugger variable values. EVTVARS	■	
group	Displays/selects active event group. GROUP [*n*]	■	
siga_mux	Selects IRQ (0, 1, 7) or External LSA Bits (0, 1, 7). SIGA_MUX [IRQ | LSA]	■	
sigb_mux	Selects IRQ (2:6), RVS*, DP(0:3), LSA bits (2:6), or WP. SIGB_MUX [LSA|WP|PCMCIA|DP|RSV|IRQ]	■	
state	Displays/selects active event group. STATE [*n*]	■	
when	Defines a when/then statement. when *event_expression(s)* then *action* [*,action*]	■	
whenclr	Clears when/then statements. WHENCLR [*number* [*,number* | *number..number* | ALL]	■	

Command	Description/Syntax	MWX-ICE	XHS
whendisable	Disables when/then statements. `WHENDISABLE [`*number*` [,`*number*` |` *number*`..`*number*` | ALL]`	■	
whenenable	Enables when/then statements. `WHENDISABLE [`*number*` [,`*number*` |` *number*`..`*number*` | ALL]`	■	
whenlist	Displays when/then statements. `WHENLIST [`*filename*`]`	■	

Displaying status or information

These commands write information to a window or file to monitor the status of the various features of the program, emulator, or debugger.

Command	Description/Syntax	MWX-ICE	XHS
disassemble	Displays disassembled memory (assembly mode). DISA [*address*]	■	■
down	Moves down specified number of stack levels. DOW [*levels*]	■	■
dump	Displays memory contents. DU [/B | /W | /L] [*address* | *address_range*]	■	■
emuvars	Displays emulator variable values. EMUVARS	■	
evtvars	Displays event system variable values. EVTVARS	■	
expand	Displays all local variables of a procedure. EXP [*stack_level*] [,*window_number*]	■	■
fopen	Opens a file or device for writing. FO [/A] [/R] *window_number*, "*filename*"	■	■
fprintf	Prints formatted output to a window or file. F *window_number*, "*format_string*" [,*argument*]...	■	■
hwconfig	Displays hardware name and version. HWCONFIG	■	
list	Displays source code. L [*line_number* | *procedure_name* | @*stack_level*]	■	■
memvars	Displays memory access variable values. MEMVARS	■	

Command	Description/Syntax	MWX-ICE	XHS
mode	Selects debugger mode (high or assembly). M [HIGH|ASSEMBLY]	■	■
monitor	Monitors variables. MON [/H | /S | /T] {*expression* | *expression_range*} [;*display_line* | ;*display_line_range* [,*display_line_range*]...]	■	■
nomonitor	Discontinues monitoring variables. NOMO [*number* | *number_range*]	■	■
printf	Prints formatted output to Command window. PRINTF "*format_string*"[,*argument*]	■	■
printsymbols	Displays symbol information. PS [/C|/D|/E|/F|/M|/R|/T|/W] [*name*[*]] [\\|\\\\|*]	■	■
printtype	Displays symbol type. PT *symbol_name*	■	■
printvalue	Prints the value of a variable. P [/H | /S | /T] {*expression* | *expression_range*}	■	■
status	Shows the status of the debugger or target. STAT [ALL | XRAY | HELP | SEARCH]	■	■
tgtmode	Displays target connection mode. TGTMODE	■	■
up	Moves up stack specified number of levels. UP [*levels*]	■	■
xicevars	Displays internal debugger variable values. XICEVARS	■	
xlate	Converts logical address to physical address. XLATE *address*	■	

Emulator configuration

These commands control the basic operation of the emulator.

Command	Description/Syntax	MWX-ICE	XHS
bclock	Enables the buffering of CLKOUT. bclock [on|off]	■	
bte	Enables/disables emulator bus timeout. BTE [ON|OFF]	■	
emuvars	Displays emulator variable values. EMUVARS	■	
isomode	Enables/disables isolation mode. ISOMODE [ON|OFF]	■	
poreset	Resets the processor with a power-on reset. poreset	■	
reset	Resets PC and synchronizes with target RESET. RESET	■	■
rte	Enables/disables realtime enforcement. RTE [ON|OFF]	■	
run_poll	Sets number of polls per second during run. RUN_POLL [*n*]	■	
run_time	Sets maximum run time before forcing a break. RUN_TIME [*n*]	■	
sit	Stops in target loop address. SIT [*address*]	■	
sitstate	Selects emulation stop method. SITSTATE [ON|OFF]	■	

Diagnostics

These commands provide a variety of diagnostic routines for target testing.

Command	Description/Syntax	MWX-ICE	XHS
diag 0	Simple target ram test. DIAG 0,*address_range* [*#count*]	■	
diag 1	Complex target ram test. DIAG 1,*address_range* [*#count*]	■	
diag 2	Continuous reads from target memory. DIAG 2,*address_range*	■	
diag 3	Continuous writes to target memory. DIAG 3,*address_range=data*	■	
diag 4	Writes alternating pattern to target memory. DIAG 4,*address_range=data* [=*alt_data*]	■	
diag 5	Continuous writes of rotating data value to target memory. DIAG 5,*address_range=data*	■	
diag 6	Continuous writes with read-after-write. DIAG 6,*address_range=data*	■	
diag 7	Writes incrementing value to target memory. DIAG 7,*address_range*	■	
diag 8	Sends continuous stream of reset pulses. DIAG 8	■	

Using macros

The macro commands define and display macros.

Command	Description/Syntax	MWX-ICE	XHS
breakaccess	Sets an access breakpoint. BA [address | address_range] [;macro_name()]	■	■
breakcomplex	Attaches a macro to an event system break. BC n ;macro_name()	■	■
breakinstruction	Sets an instruction breakpoint. BI [address | address_range] [;macro_name()]	■	■
breakread	Sets a read access breakpoint. BR [address | address_range] [;macro_name()]	■	■
breakwrite	Sets a write access breakpoint. BW [address | address_range] [;macro_name()]	■	■

define	Creates a macro.
	def [macro_type] macro_name
	([parameter_list])
	[param_definitions]
	{
	macro_body
	}
		■	■

error	Sets command file error handling. error=[quit | abort | continue]	■	■
include	Reads in and processes a command file. include "filename"	■	■
show	Displays the macro source. show macro_name() [,window_number]	■	■
vmacro	Attaches a macro to a user-defined window. vmacro window_number [,macro_name()]	■	■

Symbol and expression commands

These commands add, remove, and display symbols and expressions.

Command	Description/Syntax	MWX-ICE	XHS
add	Creates a symbol. `ADD [type] symbol_name [&address] [=value [,value]...]`	■	■
browse	Displays class inheritance information. `BROWSE symbol_name`	■	■
cexpression	Calculates the value of an expression. `C expression`	■	■
context	Shows current context. `CONT [/F]`	■	■
delete	Deletes a symbol from the symbol table. `DEL { symbol_name | \\ | \ } [,y]`	■	■
expand	Displays all local variables of a procedure. `EXP [stack_level] [, window_number]`	■	■
printsymbols	Displays symbol, type, and address. `PS [/C|/D|/E|/F|/M|/R|/T|/W] [name[*]] [\\|\\\|*]`	■	■
printtype	Displays high-level symbol information. `PT symbol_name`	■	■
printvalue	Prints the value of a variable. `P [/H | /S | /T] {expression | expression_range}`	■	■
scope	Specifies current module and procedure scope. `SCOPE [/F | root_name\\ | [root_name\\] module_name | [[root_name\\] module_name\] {procedure_name | (expression) | @stack_level | #line_number}]`	■	■

Simulating port I/O and interrupts

These commands enable simulation of interrupts and of input and output to and from port.

Command	Description/Syntax	MWX-ICE	XHS
din	Displays input port buffer values. `DIN [`*port_addr* `|` *port_addr_range*`]`	■	■
dout	Displays output port buffer values. `DO [`*port_address* `|` *port_address_range*`]`	■	■
inport	Sets or alters input port status. `INP [/B | /W | /L]` *port_address* `[,`*input_source*`]`	■	■
outport	Sets or alters output port status. `OU [/B | /W | /L]` *port_address* `[,`*output_destination*`]`	■	■
pause	Pauses simulation for specified seconds. `PA [`*n*`]`	■	■
rin	Rewinds input file associated with input port. `RI` *port_address*	■	■
rout	Rewinds output file associated with output port. `ROU` *port_address*	■	■

Utility commands

The utility commands perform miscellaneous operations.

Command	Description/Syntax	MWX-ICE	XHS
alias	Substitutes XRAY command name with another. `AL [`*`alias_name`*` [= [`*`definition`*`]]]`	■	■
status	Shows the status of the debugger or target. `STAT [ALL	XRAY	HELP	SEARCH]`	■	■
xlate	Converts logical address to physical address. `XLATE` *`address`*	■	

Connection and configuration commands

These commands provide an alternative way of defining and saving emulator connections and configurations.

Command	Description/Syntax	MWX-ICE	XHS
conclear	Clears all defined connections. CONC	■	
condelete	Deletes a defined connection. COND *symbolic_name*	■	
config	Configures a connection. CONF [*symbolic_name*, ETHERNET, *hostname*]	■	
conlist	Lists all defined connections. CONL	■	
connect	Connects to the specified emulator connection. CONN [/FO] *symbolic_name*	■	
consave	Saves the emulator connection and configuration. CONS [/A | /O] [*filename*]	■	
disconnect	Disconnects from the emulator. DISC	■	

XRAY commands not supported in MWX-ICE

The following commands are not supported in MWX-ICE.

Command
analyze
find
history
host
ice
interrupt
next
noice
nointerrupt
nomemaccess
printanalysis
printprofile
profile
ramaccess
romaccess
setstatus event
setstatus qualify
setstatus read
setstatus trace
setstatus trigger
setstatus verify
setstatus write

Command
startup
status buffer
status event
status qualify
status trace
status trigger

Chapter 10
MWX-ICE Tutorial

This chapter introduces you to the MWX-ICE debugger interface, demonstrates the use of many commands commonly used in a debug session, and provides practical examples for using the emulator in common debugging situations.

The last part of the tutorial covers the basics needed to prepare code for an embedded system application.

It is assumed that you have already set up the emulator (*Emulator Installation Guide*), installed the MWX-ICE software, and set up your environment (Chapter 2 of this manual).

Note

Before starting the tutorial, make sure that MWX-ICE is properly installed, with all its default settings, in the installation directory on your local hard disk.

How to use this tutorial

Tutorial program

The program CDEMON.ELF is used throughout the tutorial. This file was built on a Sun host, but it can be used without problems by the MWX-ICE debugger for Windows.

User-entered commands

Throughout the tutorial, an arrow in the margin indicates a procedure that you need to follow. For example, the following procedure asks you to enter the **context** command. Note that when you are asked to enter a command, you need to switch to (*activate*) to the Command window.

➤ **Example of a command for you to enter**

- In the Enter Command box, enter:

```
context
```

MWX-ICE allows most commands to be abbreviated. The abbreviated command is used whenever possible.

➤ **An example of the abbreviated form of the "context (cont)" command**

- In the Enter Command box, enter:

```
cont
```

Many of the commands used in the tutorial can also be executed by activating debugger command buttons, menus, and notebooks. These alternate methods will be noted in brackets following the command line syntax.

All the MWX-ICE commands used in the tutorial are covered in depth in Help. A command quick reference is included in Chapter 9 of this manual.

MWX-ICE debugger

General description

The MWX-ICE debugger has a windowed user interface. Commands can be entered in a variety of ways: by typing the command in a Command window, by clicking a button that activates the command, by choosing a command from the menu bar, or by choosing a command from the shortcut menus that appear in specific windows. You can also use the MWX-ICE notebooks. The notebooks provide a convenient way of entering commands: choose the notebook page for the task you want to accomplish, and fill in the blanks.

Users may find the more intuitive buttons easier to use while learning the debugger. If a button has a command line associated with it, that line will be printed in the Command window when the button is clicked. This shortens the learning curve for users who may ultimately prefer the speed of the command line by providing learning reinforcement with each button click.

Information about navigating the user interface can be found in Help.

What you need to run the tutorial

Before you can run this tutorial, you need to follow the procedures in Chapter 2. You need to define an emulator connection, connect to it, and then save your configuration. Also, the first time you connect to the emulator be sure to select the Force OS Download option (Connections window). This ensures the emulator control software is correct for the version of the debugger you are using.

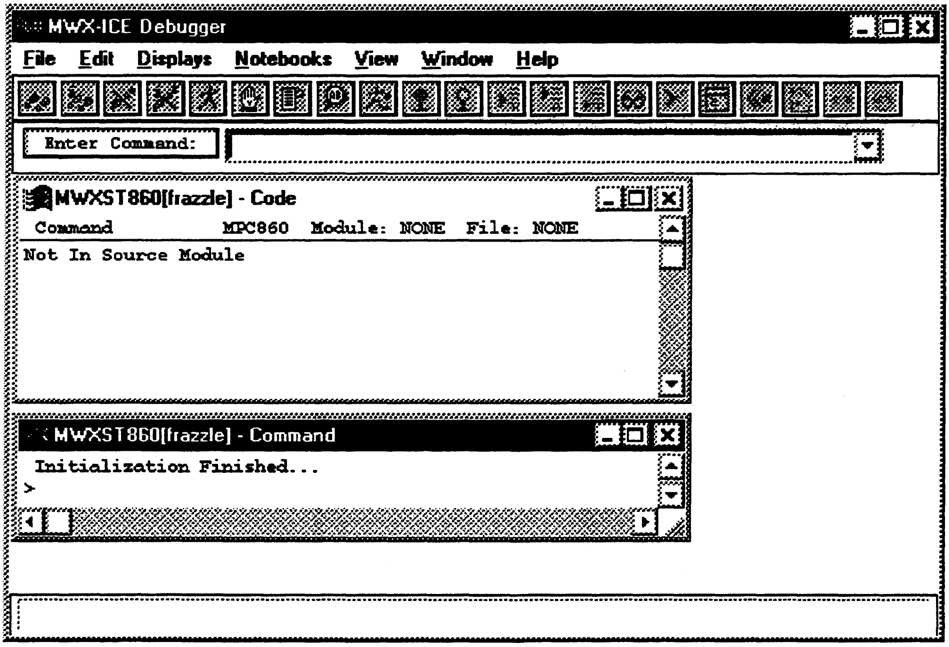
Environment variables

If your software was installed in any directory other than C:\ST8XX, make sure the XRAYMASTER environment variable has been set up before you begin this tutorial. To check the path and the environment variables you can type **set** from the DOS prompt. If XRAYMASTER has not been set up or you are not sure it is correct, please refer to Chapter 2 and set it now.

Starting the debugger

➤ To start MWX-ICE

- See Chapter 2 for startup procedures.


MWX-ICE Debugger
File Edit Displays Notebooks View Window Help
Enter Command:
MWXST860[frazzle] - Code
MWXST860[frazzle] - Command

```
Command              MPC860   Module: NONE   File: NONE
Not In Source Module
```

```
Initialization Finished...
>
```

Figure 10-1 MWX-ICE successfully connected to the SuperTAP emulator

Debugger basics

The following sections demonstrate how to:

- □ Navigate the user interface.
- □ Execute commands.
- □ Shift between Run mode and Pause mode.
- □ Exit the debugger.

General description

The MWX-ICE debugger has a windowed user interface. Commands can be entered in a variety of ways: by typing the command in a Command window, by clicking a button that activates the command, by choosing a command from the menu bar, or by choosing a command from the shortcut menus that appear in specific windows. Also available for command execution and data entry are interactive debugger Windows and Notebooks accessed through pull-down menus.

Users may find the more intuitive buttons easier to use while learning the debugger. If a button has a command line associated with it, that line will be printed in the Command window when the button is clicked. This shortens the learning curve for users who may ultimately prefer the speed of the command line by providing learning reinforcement with each button click.

Information about navigating the user interface can be found in Help.

Navigating the user interface

The MWX-ICE debugger has a multi-windowed graphical user interface (GUI). The main window can display several other windows at once — such as Command, Code, or Registers — but only one window at a time can be active.

Using the mouse

Left mouse button—Activates windows, selects text (point and drag), and invokes commands and global menus. Select items such as file names and directories by double-clicking them.

Right mouse button—Activates pop-up menus for the current window.

Selecting windows

A window becomes active when you move the mouse cursor into it and left click once. After that, any time the cursor enters the window it will become active and a region of the window that accepts keyboard input will be highlighted.

Resizing windows

You can resize a window by moving the mouse to a corner of the window, then dragging the corner until the window is the desired size.

To make a window full-size, click the Maximize button in the upper-right corner, or click the Control menu button and choose Maximize.

Scrolling windows

You can scroll up and down in an active window by left-clicking the scroll bar's up or down arrows or by clicking the scroll bar's slider then dragging it to the desired position.

Note

If you are new to Microsoft Windows 95, please run Microsoft's excellent tutorial on using Windows standard time-saving features before you continue this chapter. It takes only a few minutes, and it will make you productive much faster. In the Program Manager menu bar, click Help and then click Windows Tutorial.

Window terms and objects

Menus

Menus are lists of commands or other items you can choose. Menus are displayed by clicking on the menu name with the left mouse button. Pull-down menus always display the menu name. Option menus display the current selection.

Buttons

Go button

A command button is a small box with a word or graphic inside of it on a window. Clicking on the button performs a command.

Text fields

A text field is a long, rectangular box in which you can type text. To enter text in a field, click the left mouse button inside the field. A black border should surround the field, and a cursor should appear inside of it. If a cursor does not appear, then this field is not an input field.

Icons

An icon is a graphical representation of minimized element. Clicking on an icon displays the element.

Dialogs

A dialog is a screen that displays options for you to select. It contains command buttons that let you perform actions as well. A dialog usually opens as a result of a command, performs one specific function, and closes. For example, the box that appears when you select **Open** from the **File** menu is a dialog.

Dialogs such as the one that appears when you select **Open** from the **File** menu require an action before you can return to the window. Dialogs such as the File Chooser (described in the following section) may remain open while you perform a different task in the window.

Shortcut menus

Context-sensitive shortcut menus appear at the position of the pointer when you right-click the client area of certain windows. The commands available on a shortcut menu are duplicates of the commands on menu bar menus. Which ones appear depends on where the pointer is when you right-click. The exercises in this tutorial point you to frequently used shortcut menus.

Notebooks

Notebooks are sets of options grouped into "pages." The pages contain options that you can select from; some options require only a click on or off and others require that you enter a file name, a symbol name, or value. Each page in a notebook acts separately and must be "applied" for the selections made on that page. The pages retain your modifications as you browse through the notebook, but the options are not added to the project or tool until you click apply for each page.

Note

Notebooks are really separate programs. Therefore, if you have a notebook open and then you click anywhere in the debugger client area, the notebook seems to disappear. When it does, it is actually still open and running behind the debugger main window. To display the notebook again, press ALT-TAB.

File Chooser button

File/Directory Chooser

The File Chooser button appears whenever you need to enter a file name or directory. To open the File Chooser, click the File Chooser button and select an option from the menu that appears. Previous choices given for the file name or directory appear at the bottom of this menu in addition to the File Chooser option.

Executing debugger commands

Debugger commands can be entered in a variety of ways. You can:

- Click a toolbar button that activates the command.
- Choose a command from a menu on the menu bar.
- Enter commands and options in a notebook.

- ❑ Choose a command from one of the shortcut menus that appear in specific windows.
- ❑ Type the command in the info bar Enter Command box (but only when the Command window is active).

The contents of menu bar menus, shortcut menus, and the info bar are context sensitive; they change, depending on which window is active. For example, the Enter Command edit box is visible only when the Command window is active.

Some users naturally gravitate to the menus in the beginning. To use the menus to enter a command, click the menu you want and then click the command you want.

You might also find the buttons on the tool bar easy to use while learning the debugger. Notice that, if a button has a command line associated with it, that line is printed in Command window when the button is clicked. Although using the tool bar can shorten your learning curve, you might later prefer the speed of the Enter Command box. If you have noted the commands associate with the buttons, you will be better prepared to use the command entry box.

➤ To enter a command through a notebook

1. From the Notebooks menu, choose the notebook for the task you want to accomplish.
2. Click a tab on the notebook to select a page.
3. Fill in the blanks and select the options you want.
4. Click the command button for that page.

➤ To enter a command from a shortcut menu

1. Click the cursor in a window, a dialog box, or on a field.
2. Click the alternate (usually the right) mouse button.

   A small menu pops up, displaying commands that relate to the object you are working with in the window.

Most of the exercises in this tutorial suggest you enter commands through the Enter Command box. Look for the Enter Command box in the info box, just below the tool bar in the MWX-ICE window. The Command window needs to be active for the Enter Command box to be displayed.

➤ To execute a command using the Enter Command box

1. With the insertion point in the box, type the command.

   — or —

   Click the button to the right of the box. A dropdown list of most recently used commands appears. Select the command you want.

2. Press Enter.

   — or —

   Click the Enter Command button, to the left of the entry box.

Working in pause mode or run mode

The SuperTAP emulator operates in one of two modes: run mode, when the emulator is executing target code, and pause mode, when the emulator is not executing target code. In run mode the cursor turns into an animated icon—a "running man."

➤ To start run mode

- On the toolbar, click Go.

Go button

—or—

- In the Enter Command box, type:
  `go`

➤ To return to pause mode

- On the toolbar, click Stop.

Stop button

Exiting the debugger

You can exit the debugger at any time. There are two ways to exit MWX-ICE:

- ❑ From the Command window, type **q y** in the Enter Command box.

  —or—

- ❑ From the File menu, choose Exit Debugger.

Getting debugger Help

MWX-ICE Help feature is a context-sensitive, hypertext linked compilation of procedures, reference, and practical tips. It can be started from any debugger window.

Leave Help running while you go through the demonstration. You can minimize the application to conserve screen space and restore it when needed.

➤ **To start MWX-ICE Help**

- ■ From the menu bar, choose Help.

Typical debugging operations

About the demonstration code

Cdemon is the Applied Microsystems standard C-language demonstration program, providing examples of many code and data constructions used by C programmers. Cdemon is composed of four major functions running in main(): initial(), step(), data(), and run(). An Inspector window can be used to see the output of some of the functions.

The demonstration program is designed to be used without a target. The SuperTAP emulator runs in isolation mode, and the program uses the emulator's overlay memory. The Cdemon include file described below maps overlay memory for you, and the emulator automatically enters isolation mode when it is not connected to a target.

For a detailed explanation of the Cdemon demonstration program, see Appendix C of this manual.

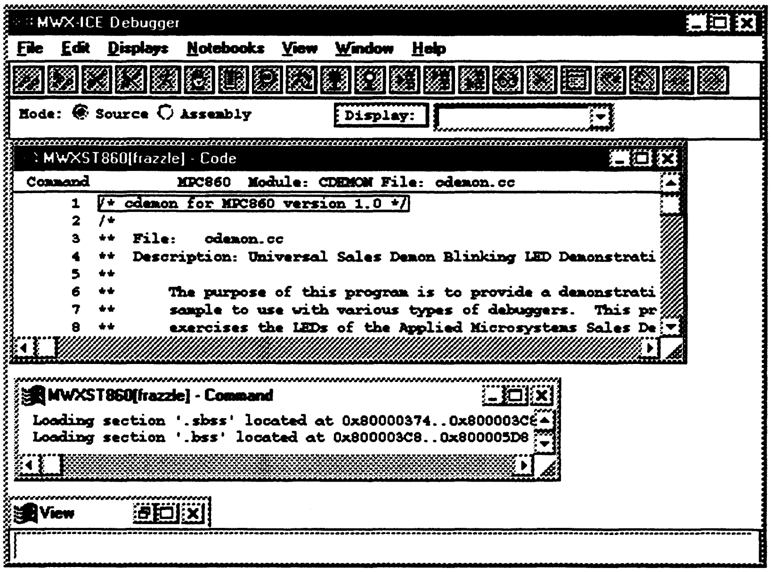
Loading the demonstration code

An include file is simply a file containing debugger commands that will be executed when the file is loaded by the debugger. The supplied include file, CDEMON.INC, maps overlay then loads the Cdemon absolute file from the DEMO subdirectory.

➤ **Load the demonstration file**

1. From the File menu, choose Change Directory.
2. Select the installation demo directory (C:\ST8XX\DEMO) and click OK.
3. From the File menu, choose Include Commands.
4. Select CDEMON.INC, and choose OK.

   Observe the Command window for SuperTAP status and executed commands. As MWX-ICE reads the include file, the commands are executed. The overlay memory map is displayed in the View Window. The Cdemon program is downloaded to the emulator's overlay memory. After the file is downloaded, the Code window displays the source module, shown in Figure 10-2. Once the code is downloaded, you can go ahead and close the View window.



Mode: ● Source ○ Assembly

```
Command              MPC860  Module: CDEMON File: cdemon.cc
/* cdemon for MPC860 version 1.0 */
/*
**  File:   cdemon.cc
**  Description: Universal Sales Demon Blinking LED Demonstrati
**
**      The purpose of this program is to provide a demonstrati
**      sample to use with various types of debuggers.  This pr
**      exercises the LEDs of the Applied Microsystems Sales De
```

```
Loading section '.sbss' located at 0x80000374..0x800003C8
Loading section '.bss' located at 0x800003C8..0x800005D8
```

Figure 10-2 Cdemon demonstration file successfully loaded

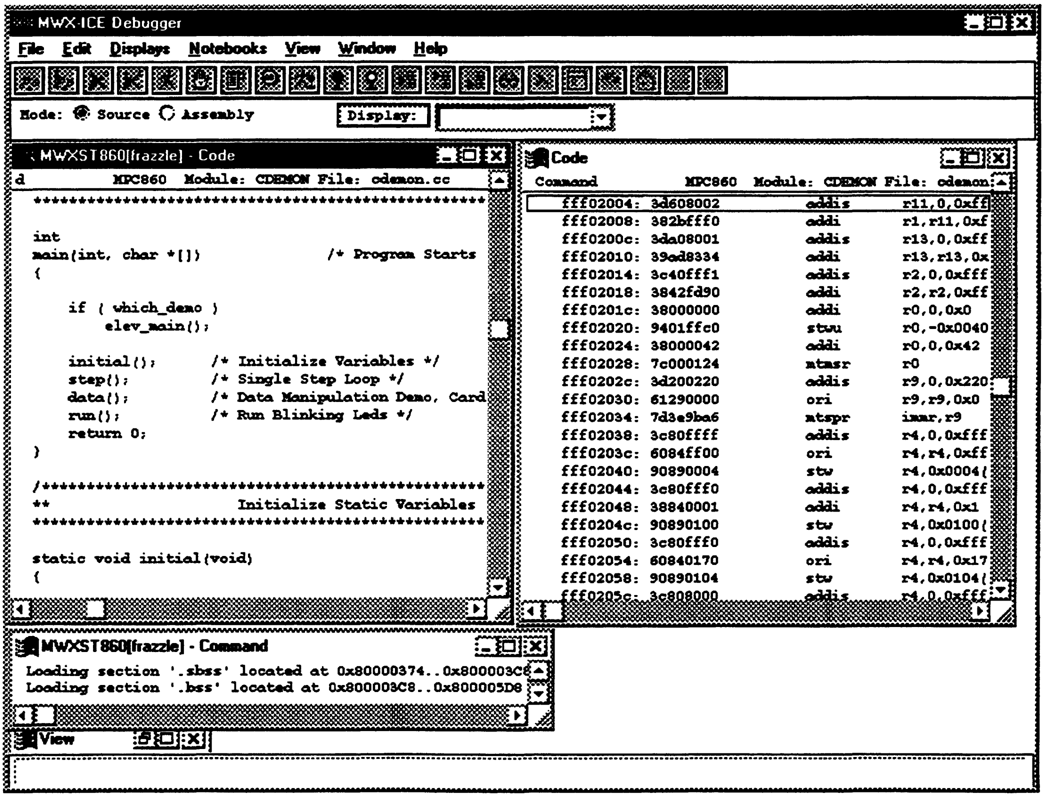
Viewing source-level and assembly code simultaneously

You can choose the Copy This Window command from the Displays menu to open multiple instances of a window. In the following example, you open two code windows: one window displaying source-level and the other displaying assembly. The two windows will stay synchronized during all emulator operations such as single-stepping, running to breakpoints, and restarting the code.

➤ Open another Code window and display assembly

1. Switch to the Code window.
1. From the Displays menu, choose Copy This Window.
2. In Mode box of the new Code window, choose Assembly.

   You can arrange the two Code windows for the best viewing, shown side-by-side in Figure 10-3.



Mode: ● Source ○ Assembly

```
d              MPC860  Module: CDEMON File: cdemon.cc
 ****************************************************

 int
 main(int, char *[])                     /* Program Starts
 {

     if ( which_demo )
         elev_main();

     initial();         /* Initialize Variables */
     step();            /* Single Step Loop */
     data();            /* Data Manipulation Demo, Card
     run();             /* Run Blinking Leds */
     return 0;
 }

 /****************************************************
 **                          Initialize Static Variables
 ****************************************************

 static void initial(void)
 {
```

```
Command                MPC860  Module: CDEMON File: cdemon
     fff02004: 3d608002            addis     r11,0,0xff
     fff02008: 382bfff0            addi      r1,r11,0xf
     fff0200c: 3da08001            addis     r13,0,0xff
     fff02010: 39ad8334            addi      r13,r13,0x
     fff02014: 3c40fff1            addis     r2,0,0xfff
     fff02018: 3842fd90            addi      r2,r2,0xff
     fff0201c: 38000000            addi      r0,0,0x0
     fff02020: 9401ffc0            stwu      r0,-0x0040
     fff02024: 38000042            addi      r0,0,0x42
     fff02028: 7c000124            mtmsr     r0
     fff0202c: 3d200220            addis     r9,0,0x220
     fff02030: 61290000            ori       r9,r9,0x0
     fff02034: 7d3e9ba6            mtspr     immr,r9
     fff02038: 3c80ffff            addis     r4,0,0xfff
     fff0203c: 6084ff00            ori       r4,r4,0xff
     fff02040: 90890004            stw       r4,0x0004(
     fff02044: 3c80fff0            addis     r4,0,0xfff
     fff02048: 38840001            addi      r4,r4,0x1
     fff0204c: 90890100            stw       r4,0x0100(
     fff02050: 3c80fff0            addis     r4,0,0xfff
     fff02054: 60840170            ori       r4,r4,0x17
     fff02058: 90890104            stw       r4,0x0104(
     fff0205c: 3c808000            addis     r4,0,0xfff
```

```
Loading section '.sbss' located at 0x80000374..0x800003C8
Loading section '.bss' located at 0x800003C8..0x800005D8
```

Figure 10-3 Displaying source-level and assembly code

Displaying configuration information

If you ever need to call Customer Support for assistance, the information displayed in the Command window during startup and the data returned by two debugger commands can help them resolve your call.

If you cannot get past startup, record any messages that appear in the Command window.

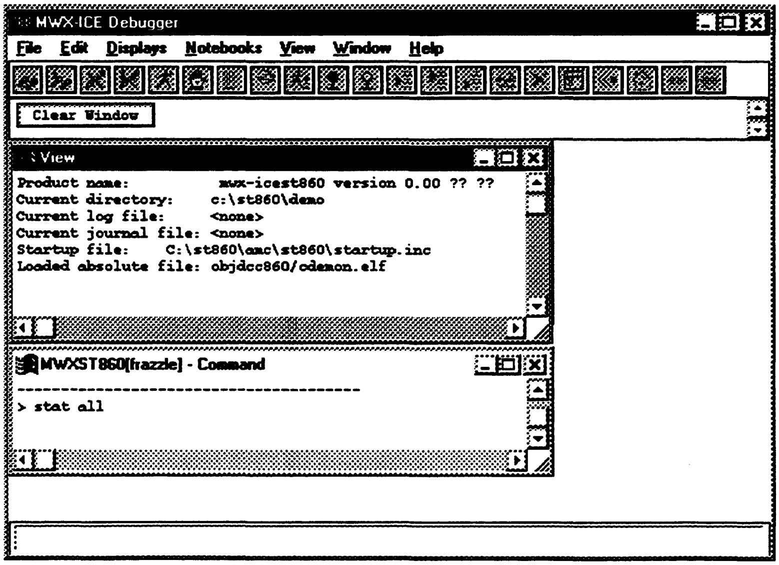
➤ To display operating environment information

1. In the Enter Command box of the Command window, enter:

   ```
   stat all
   ```

   The View window lists details about the current operating environment, as shown in Figure 10-4. If you have the View window minimized as an icon, you'll need to double-click it.

2. When you are done, close the View window (CTRL + F4).



```
Product name:          mwx-icest860 version 0.00 ?? ??
Current directory:     c:\st860\demo
Current log file:      <none>
Current journal file: <none>
Startup file:     C:\st860\amc\st860\startup.inc
Loaded absolute file: objdcc860/odemon.elf
```

```
--------------------------------------
> stat all
```

Figure 10-4 The "stat all" View window

➤ To display emulator and debugger configuration information

- In the Enter Command box of the Command window, enter:

  ```
  hwconfig
  ```

  This lists emulator component, firmware, and software revision levels and installed options in the Command window.

Modifying and saving debugger startup options and windows

You can change your MWX-ICE and emulator configuration options and then save them to the startup include file (STARTUP.INC). MWX-ICE uses the startup include file as the default configuration each time you start the debugger.

STARTUP.INC contains definitions of valid devices, a connection command, and an include-file set of configuration commands derived from the emulator configuration windows. You can have several, uniquely named startup files to configure the debugger for specific situations.

Configuration dialogs are provided for

- Defining emulator connections
- Execution control
- Trace system
- Memory configuration
- Event system setup
- File handling
- MWX-ICE interface options

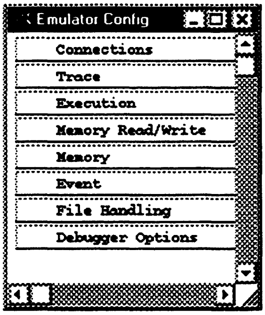


Figure 10-5 Emulator Configuration window

Displaying current configuration

➤ To display the current MWX-ICE settings

- Open all the configuration windows.

  —or—

- Execute **emuvars**, **memvars**, **xicevars**, **evtvars**, and **options** from the Command window.

Saving the current configuration

➤ To save the current debugger options

- In the Enter Command box, enter

  ```
  consave
  ```

  —or—

- From the File menu of the Emulator Configuration window. choose Save to File.

  This saves the emulator's current configuration either to STARTUP.INC or to the file name that you specify.

For detailed descriptions of all the configuration options, see Help.

Saving window position and fonts

➤ **To save the startup position of any MWX-ICE window**

- Use the Save Configuration command from the File menu.

The Save Configuration command saves the window position to a file called MASTER.INI. You can edit this file by using any text editor.

To modify window color, fonts, or mouse configuration, use the Windows Control panel.

Recording and replaying a debug session

Sometimes it may be useful to record the commands used during a debug session.

Recording commands

Sometimes it may be useful to record the commands used during a debug session. The **log** command (the Log page in the Debugger Files notebook) opens a file and saves the command line input into the named file.

➤ **To record commands and save them to a file**

- In the Enter Command box, enter:

  ```
  log on=filename.log
  ```

  where FILENAME.LOG is the name of the file you want to create.

➤ **To stop recording**

- In the Enter Command box, enter:

  ```
  log off
  ```

  The debugger stops recording and closes the log file.

Replaying commands

You can use the log file as an include file, which the debugger loads and executes to recreate a previous debugging session. You can load and execute a log file several ways:

➤ To use the command line

- In the Enter Command box, type:

  ```
  include filename.log
  ```

➤ To use the notebook

1. In the Debugger File notebook, Include page, Filename box, type:

   ```
   filename.log
   ```

   —or—

   Click the File Chooser button and choose a file from the box.
2. In the Specify Handling Mode group box, select one of the three handling modes.
3. Click the Include button.

Recording commands and their output

The **journal** command [Journal page in Debugger Files notebook] records both the commands and their output into a file. This will be demonstrated later in the tutorial where we use the command to save the contents of the emulator's "bus cycle trace" memory.

Convenience features

Command history

You can display a list of executed commands or recall a specific command from the list by clicking on the history icon located directly to the right of the Enter Command text box.

Command aliasing

The **alias** command [Alias page in the Symbol Management notebook] lets you to assign a different name to a debugger command. Preferred aliases can be placed in an "include" file and loaded at the start of your debug session

For example, the following command creates the alias **ld** for the **load** command.

```
alias load=ld
```

Getting oriented with the code

When starting a debug session you will want to get oriented with the code, particularly if the code is not your own. The following commands will help you do this.

Displaying available modules

A quick display of the names of the source modules available for debugging is a good place to start. The **printsymbols** [Info page in the Symbol Management notebook] is an important and versatile command with many options for displaying symbols and subsets of symbols. Use the **printsymbols** (**ps**) command with the **/m** flag and * argument to display all module names. Of course with very large programs containing many modules, this may be impractical.

The command will display the names of Cdemon's modules along with "type" and address information for each module. You can resize the Command window so all the options are viewable.

➤ **To display module names**

- In the Enter Command box, enter:

```
ps /m *
```

Current viewing (scope) and execution context

The debugger is capable of viewing a module that is not the current execution module. The current execution module is the module that the program counter (PC) is focused on. If you were to execute a **step** command the debugger would execute the source line pointed to by the PC. Use the **context** (**cont**) command to display the current "viewing" and "execution" modules.

➤ To display context (current viewing and execution modules)

- In the Enter Command box, enter:

```
cont
```

Note the current viewing module line, CDEMON, will have "(view)" at the end of it, while the current execution module line, ALIB, will end in "(PC)".

Changing scope

The viewing context can be changed by using the **scope** (**sc**) command [Scope button in the Code window]. This will cause the source for the module to be displayed in the Code window. It also allows access to the module's symbols and line numbers without having to type the qualifying module or procedure name, saving a considerable amount of typing.

The **scope** command is case sensitive.

➤ To change the current scope to the module DATA

- In the Enter Command box, enter:

```
sc DATA
```

➤ To display the current context

- In the Enter Command box, enter:

```
cont
```

Notice the current viewing module is now DATA, while ALIB remains the current execution module address.

➤ To return to scoping the current execution module CDEMON

- In the Enter Command box, enter:

```
sc
```

Other C source operations

There are other debugger commands that display source without changing scope:

Source code operation	Command
Display source without changing scope.	**list** [List page in the Debugger Files notebook]
Evaluate expressions.	**cexpression**
Display parameters passed to procedures.	**expand** [Stack page in the Memory Command notebook]

These commands are covered in Help.

Checking the state of the debugger and emulator

When starting a debug session you should take a quick look at the state of the debugger and emulator. This is particularly true if someone else has used the emulator between your sessions. Also, you should examine the state of the debugger and emulator any time you get unexpected results from breakpoints or event system setups.

The following commands will allow you to view and modify the parameters that control the state of the debugger and emulator.

➤ To view the emulator configuration

- From the Displays menu, choose Emulator Configuration.

  The Emulator Configuration window appears.

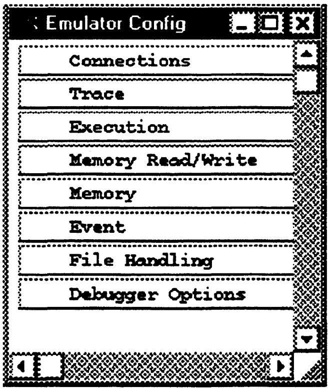


Figure 10-6 Emulator Configuration window

You can use the Emulator Configuration window to view and modify the options that control the state of the debugger and emulator:

Connections The Connections button brings up the Connections window. Use this window to define and connect to emulators.

Event The Event button opens the Event configuration dialog box. Use this dialog box to set the emulator event system options.

Execution The Execution button opens the Execution configuration dialog box. Use this dialog box to set the emulator execution options.

File Handling The File Handling button opens the File Handling dialog box. Use this dialog box to specify the upload and download format for non-IEEE-695 object files, and to other download options.

Memory Read/Write The Memory Read/Write button opens the Memory Read/Write dialog box. Use this dialog box to enable overlay memory, and to conrol access to overlay by external bus masters (Isolation of overlay read/write).

Memory The Memory button opens the Memory configuration dialog box. Use this dialog box to set memory access attributes.

Trace The Trace button opens the Trace configuration dialog box. Use this dialog box to set emulator trace collection and display options.

Debugger Options The Debugger Options button opens the Debugger configuration dialog box. Use this dialog box to set input and output radix and other debugger options

Controlling the processor and the emulator

Changing the contents of a CPU register

While debugging your code, you may find a register holding a different value than what you expected. The Register window allows you to directly modify the contents of a CPU register. This lets you replace the questionable value with the expected value and test the results.

➤ **To change the contents of a CPU register**

1. From the Displays menu, choose Register.
2. From the Register menu, choose General registers.

   The General registers window appears.
3. Click the IP (instruction pointer) button.

   The Prompt Dialog box appears and shows the current value of the counter.
4. Delete the current value; type **0x100** in the dialog box, and choose Set.

   Note the IP register value displayed in the Register window has changed to 00000100.

5. Now, set the contents of the IP back to 0xfff02004.

Note

You can also use the **setreg** command in the Command window to change the contents of registers. For example, you could use the commands: **setreg @ip=0x100** and **setreg @ip=0xfff02004** to change the contents of the IP to 0x100 and back again.

Other emulator controls

There are commands that control the handling of bus time-outs (**bte**), show cycles (**showcycle**), timestamp resolution (**timclk**), and real-time emulation (**rte**). For more information about these controls, see Help for the Emulator Configuration window. These commands are also described in Help.

Memory control

Displaying memory addresses and variables

The **dump** (**du**) command [Memory window] displays the contents of memory at a given address or range of addresses in both hexadecimal and ASCII format.

As do most MWX-ICE commands, **dump** also accepts a symbol name as an address argument. This allows us to **dump** the contents of the tutorial's *memory mapped* output port, led_port, without recalling the port's numerical address. (Note: you can also find the address of led_port with the **printsymbols** command.)

➤ **To dump the contents of led_port**

- In the Enter Command box, enter:

```
du &led_port
```

Another way of viewing the contents of led_port is to use the **printvalue** (**p**) command [Print button in the Code and Command windows]. The **printvalue** (**p**) command displays the values of expressions according to their type. The **printsymbols** (**ps**) command will show led_port is an array of signed char, so **printvalue** will display character values found at led_port.

➤ To display symbol information about led_port

- In the Enter Command box, enter:

```
ps led_port
```

➤ To print the value at led_port

- In the Enter Command box, enter:

```
p led_port
```

You may want to keep a continuous display of a variable's value on the screen. The **monitor** (**mon**) command creates a Data window and displays the selected variable. The display is updated during every *run-to-pause* transition.

➤ Monitor the variable led_port

- In the Enter Command box, enter:

```
mon led_port
```

Modifying memory

Modify memory with the **setmem** command [Memory window]. **Setmem** has a switch for byte, word, and longword data arguments. We will use **setmem** with the longword switch, **/l**, to modify the contents of led_port, and then view led_port with the **dump** and **printvalue** commands.

➤ Set memory at led_port, then display the new contents

- In the Enter Command box, enter:

```
setmem/l &led_port="MWX-ICE"
du/l &led_port
p led_port
```

Using the single line assembler

There may be times when you need to make a small change to an assembly module, perhaps just to try something out. Use the debugger's built in line assembler to make your patch and avoid a time consuming "exit debugger, edit code, assemble and link, download, and try-it-out" debugging cycle. The line assembler is invoked with the **asm** command.

➤ To assemble a "nop" loop beginning at address 0x80000100

- In the Enter Command box, enter the following commands:

```
asm org 0x80000100
asm nop
asm nop
asm b 0x80000100
```

Using the memory disassembler

Disassemble the "nop" loop at 0x80000100. The disassembled memory is displayed in the assembly mode Code window.

➤ To disassemble memory at 0x80000100

1. In the Enter Command box, enter:

```
disa 0x80000100
```

2. Switch to the Code window and select the Assembly Mode option.

   You should see code similar to the following:

```
80000100: 60000000   ori   r0,r0,0x0
80000104: 60000000   ori   r0,r0,0x0
80000108: 4bfffff8    b       .-0x8
```

3. When you are done, select Source mode to return to the source code view.

Other memory operations

There are other memory operations that you can perform.:

Description	Command Line	Memory Commands Notebook
Fill memory with a given value	**fill**	Fill page
Copy the contents of one block of memory to another	**copy**	Copy page
Compare the contents of two memory blocks	**compare**	Comp page
Search through memory for a pattern	**search**	Search page

These commands are covered in Help.

Using overlay memory

Overlay memory is emulator memory that can replace target memory by overlaying it, or be used where target memory resources do not exist. Assigning overlay memory to address ranges and access types chosen by the user is called *mapping* overlay. The memory map can be permanently stored in an MWX-ICE include file.

Note

Overlay has a minimum granularity of 128K. If a mapping does not begin and end on a 128K boundary, the emulator automatically adjusts the mapping in both directions to the next 128K boundary and issues a warning that it has adjusted the original mapping.

Displaying the memory map

Use the **map** command without arguments to display the current overlay vs. target memory map.

➤ To display current overlay vs. target memory map

1. Switch to the Command window.
2. In the Enter Command box, enter:

   ```
   map
   ```

   In the View window, you should see a display of the type of memory (RW, RO, or WO), its address range, and how much emulator overlay memory remains. The types of memory are read-only (RO), write-only (WO), or read-write (WR).
3. Close the View window (CTRL + F4).

Mapping overlay memory as RAM

You use the **map** command with a range argument to map overlay memory as read/write memory (RAM). Memory mapped as RAM is fully accessible to the executing program and to the user.

For example, the following commands map overlay from 0x1000 to 0x2000 as RAM:

```
map 0x1000..0x2000
```

Mapping overlay memory as ROM

You use the **map** command with a range argument followed by **=ro** to map overlay memory as read only memory (ROM).

If the executing program writes to memory mapped as ROM, the writes are blocked by the emulator. However, you can still write to this memory using any debugger memory write command such as **setmem** or **fill**.

For example, the following commands map overlay from 0 to 0xfff as ROM:

```
map 0x0..0xfff=ro
```

Mapping overlay memory as write only

You use the **map** command with a range argument followed by **=wo** to map overlay memory as write only memory.

If the executing program reads memory mapped as write only, the reads are blocked by the emulator. However, the user can still read this memory using any debugger memory read command such as **dump** or **disassemble**.

For example, the following commands map overlay from 0 to 0xfff as write only:

```
map 0x0..0xfff=wo
```

Mapping overlay memory back to target memory

If you have target memory, you can use the **map** command with the **=target** argument to reassign memory to the target.

For example, the following commands return overlay memory from 0x1000 to 0x2000 to target:

```
map 0x1000..0x1fff=target
```

Copying target memory contents to overlay memory

Use the following procedure when you need to copy the contents of your target ROM or PROM into overlay memory for patching, to avoid having to burn a new ROM.

Map overlay memory over the range of the ROM. Next, set up the overlay access types: **copyfrom off** (use target as the source) and **copyto on** (use overlay as the destination). Set **ovwritethru off** to block any write operation directed at the ROM to avoid contention on the address bus. Use the **copy** command to copy the contents of target memory into overlay memory starting at the same address as the target ROM's first address.

For example, the following commands copy the contents of target memory into overlay:

```
map 0x9000..0x9fff
overlay copyfrom off
overlay copyto on
ovwritethru off
copy 0x9000..0x9fff,0x9000
```

Note that unless you have target memory in the address range this example uses (0x9000..0x9fff), the copy command will cause a machine check exception.

Basic breakpoints

When debugging code, the ability to stop code execution at any desired place is absolutely necessary. To provide this ability SuperTAP supports four types of breakpoints:

- ❑ Asynchronous breakpoint
- ❑ External breakpoints
- ❑ Software instruction-execution breakpoints
- ❑ Hardware-implemented access breakpoints

When breakpoint conditions are met, SuperTAP can perform any of the following actions:

- ❑ Stop emulation (break).
- ❑ Execute a C expression.
- ❑ Log the value of an expression in a file.
- ❑ Execute a debugger macro.

Asynchronous breakpoint capability lets you stop code execution at any time clicking any of the MWX-ICE Stop buttons.

External breakpoints allow an external trigger-in signal from the target or from a piece of test equipment, such as a logic analyzer, to cause the SuperTAP to "break" out of emulation. Or, the SuperTAP can generate a trigger-out signal to trigger a logic analyzer or storage scope. SuperTAP provides one BNC trigger input and one BNC trigger output pin to support both types of external breakpoints.

Setting and clearing instruction breakpoints

Software breakpoints replace the instructions in the target program with a special opcode that forces a specific behavior in the microprocessor. When the breakpoint occurs, SuperTAP halts execution and places the original instruction back into memory.

Software breakpoints must be located in RAM, so that the special opcode may be written to target memory.This poses no problem, since SuperTAP overlay memory can easily be used for setting breakpoints where no target RAM exists.

MWX-ICE offers a variety of methods to set and clear breakpoints. Below, you sample the various methods including:

- Using the Execution Control notebook
- Choosing the BreakI button with a source line selected
- Using the shortcut menu in the Code window
- Using the Breakpoints window

➤ To set a breakpoint using a notebook

1. From the File menu, choose Restart.

   This restarts the Cdemon program.

2. From the Notebooks menu, choose Execution Control.
3. From the Execution Control menu, choose Set and clear breakpoints.
4. In the Start Address box, enter:

   ```
   main
   ```

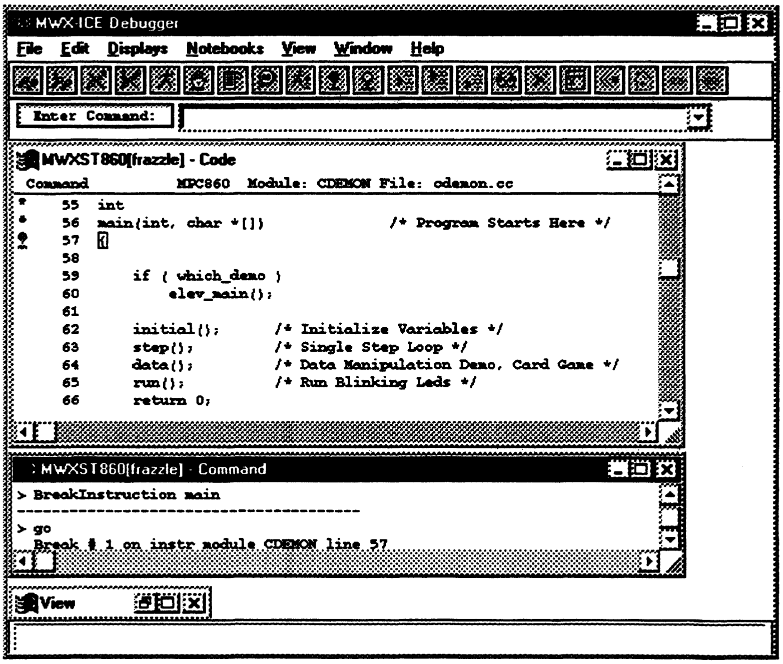
5. Choose the Set Break button, to set the breakpoint.

   Notice that “BreakInstruction main” is echoed in the Command window. By observing and remembering the commands generated while using notebooks and buttons, you can quickly start using the command line interface.

6. In the tool bar, choose the Go button to run to the breakpoint.

Notice that the Code window box indicating the current execution line has moved to the beginning of main(). Also notice the breakpoint icon (a small stop sign) in the left side of the current execution line, shown in Figure 10-7.

7. In the Execution Control notebook, choose Close, to close the window (ALT + F4).



```
Command             MPC860  Module: CDEMON File: cdemon.cc
    55  int
    56  main(int, char *[])               /* Program Starts Here */
    57  {
    58
    59      if ( which_demo )
    60          elev_main();
    61
    62      initial();      /* Initialize Variables */
    63      step();         /* Single Step Loop */
    64      data();         /* Data Manipulation Demo, Card Game */
    65      run();          /* Run Blinking Leds */
    66      return 0;
```

```
> BreakInstruction main
------------------------------------
> go
  Break # 1 on instr module CDEMON line 57
```

Figure 10-7 Breakpoint at beginning of main()

➤ To set a breakpoint using the BreakI button

1. In the Code window, select the source line containing the step() function by double-clicking anywhere in that line.

   When selected, there will be at least one character of the line highlighted.

2. To set the breakpoint, choose the BreakI button from the tool bar.

   Notice the breakpoint icon now displayed left of the step() source line.

3. Choose the Go button to run to the breakpoint.

➤ To set a breakpoint using the shortcut menu

1. In the Code window, click the to the left of the line number for the run() function using the right mouse button.

   The shortcut menu appears. Shortcut menus contain commands related to the window and to the current context.

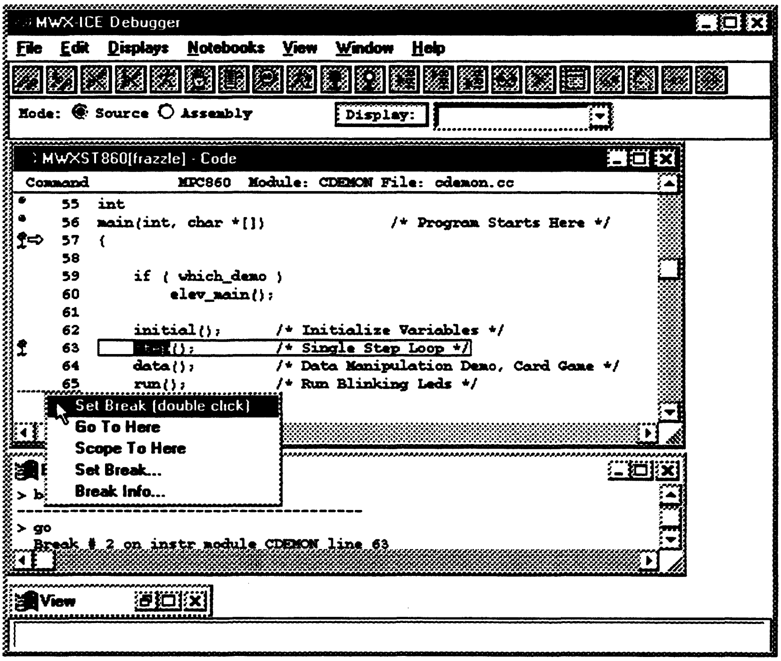
2. Choose Set Break [double click] from the shortcut menu.

   Notice the breakpoint icon now displayed left of the run() source line. You can also set a breakpoint by double-clicking to the left of the line number.

3. Choose Go, to run to the breakpoint.

Note

You can also choose Set Break... from the shortcut menu to open the Define a Breakpoint dialog box.



```
Command            MPC860  Module: CDEMON File: cdemon.cc
 55  int
 56  main(int, char *[])              /* Program Starts Here */
 57  {
 58
 59      if ( which_demo )
 60          elev_main();
 61
 62      initial();       /* Initialize Variables */
 63      step();          /* Single Step Loop */
 64      data();          /* Data Manipulation Demo, Card Game */
 65      run();           /* Run Blinking Leds */
```

```
> go
  Break # 2 on instr module CDEMON line 63
```

Figure 10-8 Setting a breakpoint using the shortcut menu

➤ View the breakpoints

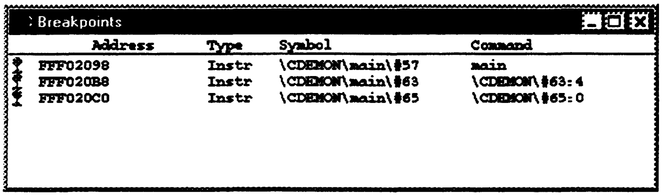
- From the Displays menu, choose Breakpoints.

  —or—

- Select Break Info from the shortcut menu. To open the shortcut menu, use the right mouse button and click to the left of the line numbers in the Code window.

  Notice the three breakpoints displayed in the window. The first one was set at main() using the Execution Control notebook. The second one was set at step() by choosing the

BreakI button with a line selected. The third was set at run() using the shortcut menu. The Breakpoints window is shown in Figure 10-9.



Address	Type	Symbol	Command
FFF02098	Instr	\CDEMON\main\#57	main
FFF020B8	Instr	\CDEMON\main\#63	\CDEMON\#63:4
FFF020C0	Instr	\CDEMON\main\#65	\CDEMON\#65:0

Figure 10-9 Displaying information about a breakpoint

➤ Clear the breakpoints

1. In the Breakpoints window, select the top line by double-clicking in that line.
2. Choose the Clear button from the tool bar.
3. Repeat steps 1 and 2 until the remaining breakpoints are cleared.

Other ways of clearing breakpoints

There are several ways you can clear breakpoints in addition to the way you cleared them from the Breakpoints window in the previous step.

From the...	Do the following...
Breakpoints window	Use the shortcut menu to clear breakpoints. Right-click on the breakpoint and choose Clear Break, or right-click in the Breakpoint window and choose Clear All. Select the breakpoint, and choose Clear Break from the View menu.
Code window	Select the source lines where the breakpoints are set, then click the Clear button. Use the shortcut menu to clear breakpoints. To open the shortcut menu, right-click to the left of the line number where the breakpoint is set. Choose Clear Break from the menu. Double-click the right mouse button to the left of the line number where the breakpoint is set.
Command window	Use the clear command. Syntax: **clear** [*n*] *n* is the breakpoint number as it is shown in the Breakpoints window. To clear all breakpoints type **clear** without a number.

Setting temporary breakpoints

Another kind of instruction breakpoint you can set is a temporary breakpoint. You can use temporary breakpoints when you want to run to a specific place in your code. These breakpoints do not appear in the Breakpoint window. You can set these breakpoints by selecting the source line you want to break on, and then clicking the GoUntil button in the tool bar.

You can also set temporary breakpoints by using the Execution Control notebook, or by using the shortcut menu in the Code window.

Setting and clearing access breakpoints

Hardware access breakpoints use the SuperTAP's hardware and do not consume any target resources. When a memory access occurs that matches the breakpoint condition, microprocessor execution stops. Access breakpoints can be set over target RAM or ROM.

Access breakpoints are set to break on an address with the following status options:

Access	Command	Notebook
Read access	**br**	Break page in Execution Control notebook
Write access	**bw**	Break page in Execution Control notebook
Read or write	**ba**	Break page in Execution Control notebook

Access breakpoints are useful for detecting errant accesses to memory locations. For example, you may want to break emulation if your code writes to a read-only port address.

Macros can be attached to access breakpoints. You may have multiple **ba**, **br**, and **bw** breakpoints set, each with its own macro attached.

In the next example, you set a write access breakpoint (**bw**) at address 0xA000000 (the address of the LEDs. The pointer dot_port holds this address. Read access (**br**) and read-or-write access (**ba**) breakpoints are set similarly.

➤ To set a write access breakpoint

1. Switch to the Command window.
2. In the Enter Command box, enter:

   ```
   restart
   ```

   This positions the program counter at the start of the Cdemon program.
3. In the Enter Command box, enter:

   ```
   bw dot_port
   ```

4. Choose Go 🏃 to run to the breakpoint.

   SuperTAP breaks emulation in the outled() function at the write to the memory-mapped address of the display dots.

➤ Clear the write access breakpoint

1. Open the Breakpoints window.
2. From the View menu, choose Show Break #.
3. Switch to the Command window, and in the Enter Command box, enter:

   ```
   clear 1
   ```

   Note: Because this was the only breakpoint set, it was number one. If it was the second of two breakpoints, you would use the command **clear 2** instead, third use **clear 3**, and so on. Use **clear** without a number to clear all breakpoints at once.
4. Close the Breakpoints window.

Program execution and related commands

The following commands control resetting the CPU, restoring the program start address, and executing the program in realtime or in steps at a time.

Resetting the processor

Use the **reset** (**rese**) command to restore the processor to its initial reset state.

➤ To reset the CPU

- In the Enter Command box, enter:

  ```
  rese
  ```

Restoring the program start address

Use the **restart** (**rest**) command to reset the program counter to the program's starting address. For CDEMON.ELF this returns us to address 0xfff02004.

➤ To restore the program start address

- In the Enter Command box, enter:

  ```
  rest
  ```

Starting and stopping program execution

Use the **go (g)** command (or the Go button) to start or continue program execution. The program runs until a breakpoint is reached, an error occurs, or the you stop emulation by clicking the Stop button ✋ from the tool bar.

Use the **go** command with an address and a passcount to execute until the address is seen "passcount" number of times. The command sets a temporary breakpoint at the address and counts each occurrence of the breakpoint. The cursor will turn into a running man indicator while the program executes.

➤ **To execute until outled() is seen four times**

1. In the Enter Command box, enter:

   ```
   g outled%%4
   ```

   After the fourth occurrence of outled(), the emulator will break and display the current instruction address at the time execution stopped.

2. Switch to the Code window and select Assembly Mode.

   After the fourth occurrence of outled(), the emulator will break and display the current instruction address at the time execution stopped.

Stepping through the program

Stepping refers to executing code a number of instructions or lines at a time. Single stepping executes either one assembly line or one source line of code at a time. To single step use the **step (s)** command (StepInstr or StepLine button from the tool bar) without a number argument.

For example, the following command executes five lines of code:

```
s 5
```

Use the **stepover (so)** command (StepOver or StepO Instr button in all windows) if you want to single step but do not want to step through called routines. This command will execute the entire called routine then stop.

Use the **gostep** (**gos**) command if you want to step continuously until a specific condition is met. The condition is defined by a macro you attach to the **gostep** instruction. For instance, **gostep** can be used to step until a register holds a particular value.

For example, the following command single-step until a condition defined in my_macro is met:

```
gos my_macro()
```

Capturing and displaying execution trace history

The trace capture feature lets the user observe exactly how the code executed. Raw trace consists of CPU bus level information including address, data, status, and timestamp information. Disassembled trace is displayed as assembly, source, or a mixture of both. Raw and disassembled trace are both displayed in the Command window.

Clearing trace memory

You may want to clear the trace memory buffer of previous trace information before running your code. This ensures all information in the trace buffer will be newly acquired. Use the **trclr** command to clear the trace memory buffer.

➤ **To clear the trace memory buffer**

- In the Enter Command box, enter:

```
trclr
```

You can also clear trace memory from the Emulator Trace window. From the Displays menu, choose Emulator Trace. From the Actions menu, choose Clear Trace.

Capturing trace in run mode

Two emulator commands, **trsys** and **trace**, control the capture of program execution trace history. With **trsys** and **trace** on, every time you use **go** or **step**, the bus information generated is captured in the trace buffer. To find out if these variables are on or off, execute the **emuvars** command and observe their status displayed in the list of emulator variables. If either variable is off, you turn it on by entering the appropriate command, **trsys on** or **trace on.**

➤ To enable the trace capture system

1. In the Enter Command box, enter:

```
trsys on
```

2. In the Enter Command box, enter:

```
trace on
```

The SuperTAP is designed to operate even when you are using the processor instruction and data caches. The SuperTAP trace system requires that the processor instruction show cycles are enabled. In most cases, you only need to generate show cycles for indirect branching. The MWX-ICE **showinst** command controls processor show cycles.

➤ To enable instruction show cycles

- In the Enter Command box, enter:

  ```
  showinst indirect
  ```

  Leave the show cycles enabled for the remainder of the tutorial.

To collect trace information, you **restart** and then **go** to the Cdemon function house().

➤ To restart, then go to house()

- In the Enter Command box, enter:

  ```
  rest
  g DATA\house
  ```

Displaying raw trace history in the Command window

Use **drt** for displaying raw bus cycle information and optional logic state and timestamp information.

➤ To display the captured raw trace information

- Maximize the Command window and in the enter the following command:

  ```
  drt
  ```

  —or—

  From the Displays menu, choose Emulator Trace.

The Frame numbers on the far left of the trace are used to reference when in trace history the information occurred. Frame 0 is the end of trace. The smaller Frame numbers are the last cycles captured prior to a "break" in emulation.

The other raw trace columns show the address (Address), data (Data), status of various CPU signals, and timestamp information for each bus cycle captured.

Searching trace history for a pattern

To search trace memory for patterns, use the **tsrch** **(ts)** command, or the Search button from the Emulator trace window.

Using the **tsrch** command, you can qualify the search with combinations of address, data, statue, and logic state analysis patterns. You can also specify a starting line number in trace history.

In the Emulator Trace window, you can use the Search button to search trace for specific text strings.

➤ To search trace history for an access to address 0x80000430

- In the Enter Command box, enter:

```
tsrch addr=0x80000430 && stat=wr
```

Displaying disassembled trace history

Use **dtb** (display trace backwards) for displaying the trace buffer information formatted in assembly or high-level mode, or as an interleaving of both modes. The **dtf** (display trace forwards) command performs the same trace display function, but in a different direction. Use the **dt** command with a start address to begin disassembling at a particular line in trace.

The **trdisp** MWX-ICE variable controls the disassembled trace display mode. The variable's default (**both**) causes an interleaving of assembly and source.

➤ To display the trace information in disassembled format

- In the Enter Command box, enter:

```
dtb
```

The numbers on the far left of the disassembled trace correspond directly to the Frame numbers on the far left of the raw trace display. They are useful when correlating a line of disassembled trace to its bus cycle equivalent line in raw trace.

Observe the call to house(). Notice the branch and link (bl) instruction in the disassembled trace display, which also shows the non-sequential changes to the IP. The previous IP is shown in the form *address* > IP. The new IP is shown in the form IP < *address*. If you look at the Code window in Assembly mode, you can see that the new IP address matches the current execution address in the Code window.

Saving trace to a file

You may need a hardcopy of trace or a copy of trace on disk for later referencing. Or, you may have a problem that requires factory support. The Applications department might request a hardcopy of trace memory to assist in solving the problem.

Earlier we discussed the **journal (jou)** command [Journal page in Debugger Files notebook], which records both the commands and their output into a file. You can use the **journal** command to save a partial or entire trace disassembly into a file. The example below shows how to save a trace memory display to a file.

For example, the following commands save part of a raw trace to a file named trace.raw:

```
jou on="trace.raw"
drt 0..42
jou off
```

The command **jou on="trace.raw"** creates a file named trace.raw as the journal file. The command **drt 0..42** displays raw traces lines 0 through 42. This display goes to both the Command window and the journal file. The command **jou off** stops recording and closes the journal file.

Using the **/a** option with **jou** allows you reopen and append to an existing file.

Capturing trace in pause mode

An emulator softswitch, **ppt**, controls the capture of additional information. With **ppt on** you can capture bus cycle information generated by MWX-ICE memory read and write commands such as **setmem**, **fill**, **copy**, **diag**, and others. This trace information can assist you in diagnosing general memory problems or memory errors that may have shown up in one of MWX-ICE's RAM diagnostic tests (**diag**).

Also, with **ppt on**, cycles generated by MWX-ICE memory commands or by downloading code with the **load** command are included in trace memory. The **load** command cycles can be a valuable source of troubleshooting information when a download fails for some reason. You can examine the last cycle in trace memory and determine if the download went to valid RAM memory, nonexistent memory, or ROM, for example.

Executing MWX-ICE commands in run mode (dynamic run mode - drun)

The **drun** command lets you use MWX-ICE commands without stopping program execution (run mode).

For instance, you may want to examine trace history (**drt** or **dtb**) while executing your program. If you enter run mode using **drun** instead of **go**, you can use the **drt** command to display the trace history, gathered up to the point where you entered the **drt** command, while the target program continues to run.

To exit dynamic run mode use the **dstop** command.

➤ To restart, enter dynamic run mode, then display raw trace history

1. In the Enter Command box, enter:

   ```
   rest
   drun
   drt
   ```

   Examine trace.

2. Restore the Command window to its previous size.

➤ To exit dynamic run mode (dstop)

- In the Enter Command box, enter:

  ```
  dstop
  ```

Timestamp

In raw trace, the timestamp information shows either the time between successive bus cycles, or the time relative to a specified trace frame number. The following MWX-ICE variables control capturing and displaying timestamp information.

timclk Selects timestamp clock resolution (40ns, 200ns, 1us, 10us, 100us, 1ms, 10ms, or 100ms).

trstamp Display timestamp interval/offset when trace is displayed in the Command window.

trbase Selects trace frame to use as timestamp base frame when trace is displayed in the Command window.

Debugger macros

Macros provide an efficient means of executing repetitive tasks or generalizing a task that originally acted on only a specific item. MWX-ICE uses the same C-like sequence of expressions, statements, and debugger commands as XRAY to define and invoke macros. See the *XRAY Debugger for Windows Reference Manual* for an explanation on how to generate your own macros and how to use the predefined macros that come with MWX-ICE. The following section demonstrates briefly how to create a macro and then save it into an "include" file that can be executed by the debugger.

Creating a macro

Use the Macro page of the Symbol Management notebook to create a macro. This puts MWX-ICE in the macro define mode.

When you use a keyboard command in a macro you must precede and follow the command with a dollar sign ($).

Below, you create a macro demonstrating using the aliased command **drt**.

➤ Define a macro named dmp_trc

1. From the Notebooks menu, choose Symbol Management.
2. From the Symbol Management menu, choose Create or edit a debugger macro.

   The Symbol Management notebook opens to the Edit Debugger Macro page.
3. In the Specify Macro Name field type:

   ```
   dmp_trc
   ```
4. Choose Edit.
5. Edit the file in the File Editor window to match the following by adding the lines shown in bold type:

   ```
   define dmp_trc()
   {
   $
   drt
   $
   ;
   }
   ```

6. From the File menu of the File Editor, choose Save, to save the macro.
7. From the Debugger Macros menu of the File Editor, choose Send Macro to Debugger.
8. From the File menu, choose Save and Exit.
9. Close the Symbol Management notebook (ALT + F4).

Displaying a macro

Use the **show** command to display an active macro in the Command window. The macro is displayed in the Command window. You may need to enlarge the window to view it. (To edit a macro, enter its name in the field of the Macro page of the Symbol Management notebook, and choose Edit.)

➤ **To display the macro dmp_trc**

- In the Enter Command box, type **sh dmp_trc**

Assigning a macro to a breakpoint

A macro can be assigned to a breakpoint by setting a breakpoint and following it with "; your_macro()". The macro is executed when the breakpoint occurs.

➤ **To assign macro "dmp_trc" to a write access breakpoint at "led_port"**

- In the Enter Command box, enter:

```
bw led_port;dmp_trc()
```

➤ **Restart, then go until the breakpoint is reached**

- In the Enter Command box, enter:

```
rest
g
```

When the breakpoint at "led_port" occurs, emulation stops, and raw trace information is immediately displayed in the Command window.

➤ To clear write breakpoint number 1

- In the Enter Command box, enter:

  ```
  cl 1
  ```

Deleting a macro

Use the **delete** command to delete a macro. For example, the following command deletes a macro called big_mac:

```
del big_mac
```

Saving a macro to an "include" file

After you determine that your macro works, you may want to save it to a file for later use. The resulting file can be used as an include file that recreates the macro.

➤ To save a macro to a file

1. From the Notebooks menu, choose Symbol Management.
2. From the Symbol Management menu, choose Create or edit a debugger macro.

   The Symbol Management notebook opens to the Edit Debugger Macro page.
3. In the Specify Macro Name field, select the name of the macro that you've created, then click the Edit button.
4. From the File menu of the File Editor, choose Save As:
5. In the dialog box, type:

   ```
   dmp_trc.inc
   ```
6. In the dialog box, choose Save, to write the macro into the file DMP_TRC.INC.

If a macro is no longer present in the Edit window but has been loaded into the debugger, you can still save it to a file.

For example, the following commands save and invoke a macro using the command line.

```
fopen 60, "dmp_trc.inc"
show dmp_trc,60
```

```
vclose 60
```

The command **fopen 60** creates a file named DMP_TRC.INC in the current directory; include a path if you want it saved elsewhere. (The number after **fopen** used must be greater than 50.) The file contains the commands necessary to create the macro **dmp_trc**, placed there by the **show** command. The command **vclose 60** saves the file. To enable the macro in future sessions, you must first invoke the include file:

```
inc dmp_trc.inc
```

Then you can use the macro.

Note

You can also define a macro in a text file, and include that file from the command line of the Command window. For correct syntax, see the examples provided for the **define** command in Help.

Using the event system

Sometimes running to a basic breakpoint and examining trace history does not provide information specific enough to debug your target's code or hardware. Also, you may want the emulator to perform some action other than breaking when the conditions become true. You may need to trigger an oscilloscope after a complex set of CPU bus cycle conditions become true, or to trace only certain types of bus cycle information under certain conditions. For example, the conditions might be the fifth write that a specific subroutine makes to a certain I/O location.

The event system can be useful when implemented as a built-in bounds checker for the executing program. Simply set up the event system to cause a break in areas of memory that nothing should access.

The event system supplies the mechanism to define conditions and execute actions through user defined "when event/then action" statements. This mechanism allows the emulator to perform various actions based on events of complexity far surpassing that of simple breakpoints.

This section will help you get started using the event system. Comprehensive user information and descriptions of all available conditions and actions are in Chapter 7, "Using the Event System."

General information

The event system is implemented with emulator hardware and can be used in both RAM and ROM regions.

Setting up when/then event statements

The first step in setting up a when/then statement is deciding what condition(s) you need to include. For most simple address and status conditions you probably need only an access breakpoint. We begin with those conditions however, to keep the first event statement simple.

➤ To define a when/then statement to cause a break on any write to "led_port"

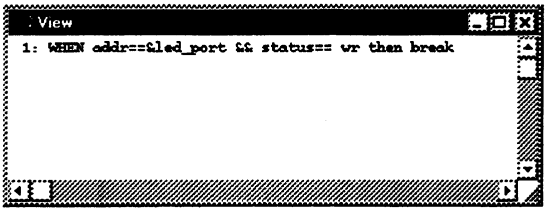
- In the Enter Command box, enter:

```
when addr==&led_port && status== wr then break
```

The command is made up of the following elements:

- ❑ **addr==&led_port** defines the address of "led_port" as the address of interest.
- ❑ **status==wr** defines the access to "led_port" as a write.
- ❑ **&&** is the logical AND operator.
- ❑ **break** stops emulation.

The event is assigned event number 1 and is displayed in the View window, shown in Figure 10-10. If you have minimized the View window as an icon, you need to open it.



```
1: WHEN addr==&led_port && status== wr then break
```

Figure 10-10 View window for viewing event system setup

➤ To run the program until the event occurs

- In the Enter Command box, enter:

```
rest
g
```

The Command window displays a message that an "Event break occurred", caused by "When statement #1." In the Code window you should see a highlighted source line indicating where the break occurred. You can scroll up to see where the write to led_port occurred.

Assigning a macro to an event system "break" action (breakcomplex)

You can use the **breakcomplex** (**bc**) command to tie a macro to an event system "break" action.

In general, to set up for a **breakcomplex**:

1. Set up the event statement.
2. Tie the macro to the event statement number using the **bc** command.

For example, the following commands attach the macro dmp_trc() to an event system trigger. This example assumes the event is event number one.

```
when addr==&led_port && stat==wr then break
bc 1; dmp_trc()
```

Additional event system features

In addition to the simple conditions and actions illustrated here, the event system possesses many advanced features such as groups, counters, timer, flags, conditional tracing, trigger generation, and others. These features are covered in Chapter 7 of this manual.

Scope loops and diagnostics

Built in scope loops and memory diagnostic programs are included with the debugger in the form of **diag** commands. These programs save you from writing your own routines to test memory or to stimulate memory for "scoping" or logic analysis.

Another diagnostic, named **crc**, calculates the CRC-16 (cyclic redundancy check) over the desired range.

Memory and IO read/write scope loops

Diagnostics 2 through 7 are used to perform reads and/or writes of selected memory with patterns chosen by the user.

➤ **To perform a continuous read**

1. In the Enter Command box, enter:

   ```
   diag 2,0x80000000..0x80000100
   ```

2. Press the Stop button to stop the test.

Memory diagnostics, simple and complex

Diagnostic numbers 0 and 1 perform simple and complex diagnostics on the selected memory.

➤ To perform a complex memory test

1. In the Enter Command box, enter:

```
diag 0,0x80000100..0x80000200
```

2. Press the Stop button to stop the test.

Cyclic redundancy check

Use the **crc** command with a range argument to perform a CRC-16 of the specified range. The command will return a hex value for the CRC.

➤ To perform a cyclic redundancy check

- In the Enter Command box, enter:

```
crc 0x80000100..0x80000200
```

Using the SuperTAP — practical examples

This section contains practical examples demonstrating using the emulator in common debugging situations including:

- Tracing a particular subroutine
- Capturing "qualified" trace history
- Capturing and viewing trace while running
- Displaying structures
- Browsing the CPU registers
- Displaying and modifying memory
- Monitoring and modifying variables while running

Example 1 – Tracing a particular subroutine

In the example, you set up trace triggering so that only the cycles within a certain subroutine run() are traced. To do this, you define a dual-event trace trigger that starts trace capture when run() is entered, and stops trace capture when run() transfers program execution.

➤ To restart the Cdemon program

- In the Enter Command box of the Command window, enter:

  ```
  restart
  ```

➤ To set up initial trace conditions

1. In the Enter Command box, enter:

   ```
   trinit off
   ```

   This keeps trace capture turned off unless enabled by the event system or the **trace on** command.

2. In the Enter Command box, enter:

   ```
   trqual dxqual
   ```

   This ensures that sufficient information is captured for the most accurate disassembly of trace.

3. In the Enter Command box, enter:

   ```
   trrunclr on
   ```

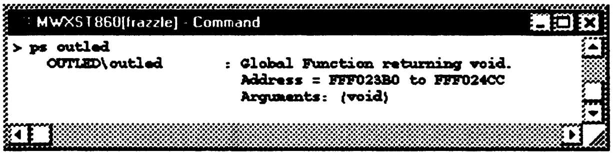
   This clears the trace buffer when the emulator enters run mode.

➤ To set up event statements to start and stop tracing

1. In the Enter Command box, enter:

   ```
   ps outled
   ```

From the output of the **printsymbols (ps)** command, we can easily determine the beginning and end address of the function outled().



```
> ps outled
    OUTLED\outled        : Global Function returning void.
                           Address = FFF023B0 to FFF024CC
                           Arguments: (void)
```

Figure 10-11 Printsymbols output for outled()

2. In the Enter Command box, enter:

   `when addr==0xFFF023B0 then tron`

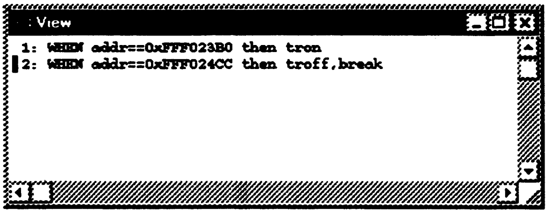
   This event statement turns tracing on at the start address of the function outled(). Notice a View window is opened when you enter the command.

3. In the Enter Command box of the Command window, enter:

   `when addr==0xFFF024CC then troff,break`

   This turns tracing off at the final address of the function outled(), then breaks emulation.

4. In the View window, examine both event system statements. See Figure 10-12.



```
 1: WHEN addr==0xFFF023B0 then tron
 2: WHEN addr==0xFFF024CC then troff,break
```

Figure 10-12 Event system "turn trace on" and "turn trace off" event statements

➤ To run until the break condition is met

- Click the Go button.

➤ To view mixed source and assembly level disassembled program execution history

1. From the Displays menu, choose Emulator Trace.

   The Emulator Trace window is displayed. It may take a few moments before the raw trace appears. See Figure 10-13.

2. From the View menu, choose Show source trace and clear Show raw trace.

   A check mark appears next to the Show option selected. Notice that only the source lines for the subroutine outled() are displayed in the trace. See Figure 10-14.

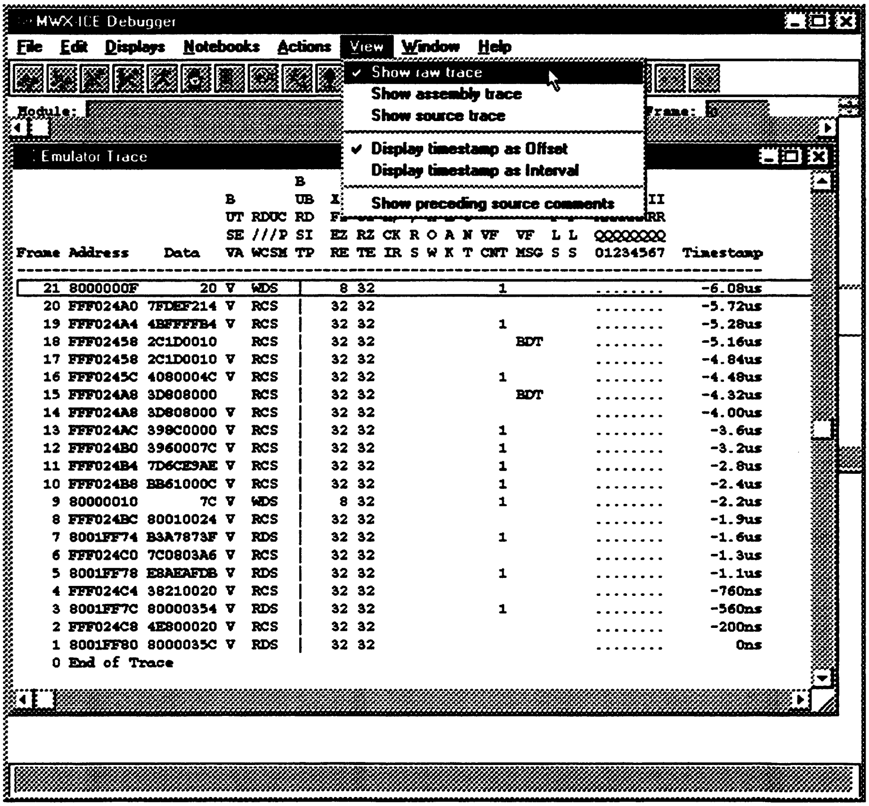
3. From the View menu, select Show assembly trace.

   Notice that only the source lines and their associated assembly code for the subroutine outled() are displayed in the trace buffer window, shown in Figure 10-15.

➤ To clear the event system

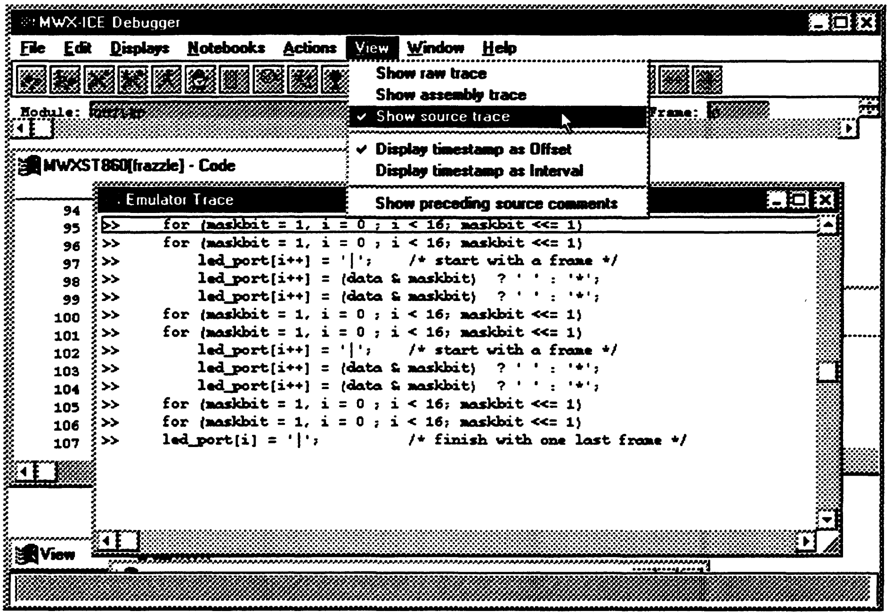
- In the Enter Command box, enter:

```
whenclr all
```



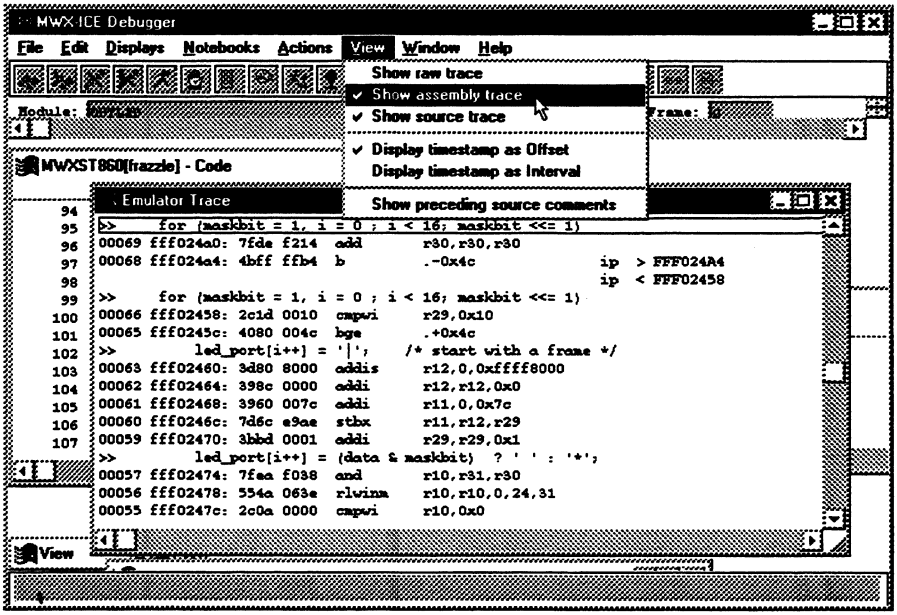
```
                                B
                        B       UB X                                   II
                        UT RDUC RD F                                   RR
                        SE ///P SI EZ RZ CK R O A N VF  VF  L L  QQQQQQQQ
Frame Address    Data   VA WCSM TP RE TE IR S W K T CNT MSG S S  01234567  Timestamp
------------------------------------------------------------------------------------
   21 8000000F       20 V  WDS  |   8 32              1          ........    -6.08us
   20 FFF024A0 7FDEF214 V  RCS  |  32 32                         ........    -5.72us
   19 FFF024A4 4BFFFFB4 V  RCS  |  32 32              1          ........    -5.28us
   18 FFF02458 2C1D0010    RCS  |  32 32                BDT      ........    -5.16us
   17 FFF02458 2C1D0010 V  RCS  |  32 32                         ........    -4.84us
   16 FFF0245C 4080004C V  RCS  |  32 32              1          ........    -4.48us
   15 FFF024A8 3D808000    RCS  |  32 32                BDT      ........    -4.32us
   14 FFF024A8 3D808000 V  RCS  |  32 32                         ........    -4.00us
   13 FFF024AC 398C0000 V  RCS  |  32 32              1          ........     -3.6us
   12 FFF024B0 3960007C V  RCS  |  32 32              1          ........     -3.2us
   11 FFF024B4 7D6CE9AE V  RCS  |  32 32              1          ........     -2.8us
   10 FFF024B8 BB61000C V  RCS  |  32 32              1          ........     -2.4us
    9 80000010       7C V  WDS  |   8 32              1          ........     -2.2us
    8 FFF024BC 80010024 V  RCS  |  32 32                         ........     -1.9us
    7 8001FF74 B3A7873F V  RDS  |  32 32              1          ........     -1.6us
    6 FFF024C0 7C0803A6 V  RCS  |  32 32                         ........     -1.3us
    5 8001FF78 E8AEAFDB V  RDS  |  32 32              1          ........     -1.1us
    4 FFF024C4 38210020 V  RCS  |  32 32                         ........     -760ns
    3 8001FF7C 80000354 V  RDS  |  32 32              1          ........     -560ns
    2 FFF024C8 4E800020 V  RCS  |  32 32                         ........     -200ns
    1 8001FF80 8000035C V  RDS  |  32 32                         ........        0ns
    0 End of Trace
```

Figure 10-13 Raw trace display of the subroutine outled()



```
>>     for (maskbit = 1, i = 0 ; i < 16; maskbit <<= 1)
>>     for (maskbit = 1, i = 0 ; i < 16; maskbit <<= 1)
>>         led_port[i++] = '|';     /* start with a frame */
>>         led_port[i++] = (data & maskbit)  ? ' ' : '*';
>>         led_port[i++] = (data & maskbit)  ? ' ' : '*';
>>     for (maskbit = 1, i = 0 ; i < 16; maskbit <<= 1)
>>     for (maskbit = 1, i = 0 ; i < 16; maskbit <<= 1)
>>         led_port[i++] = '|';     /* start with a frame */
>>         led_port[i++] = (data & maskbit)  ? ' ' : '*';
>>         led_port[i++] = (data & maskbit)  ? ' ' : '*';
>>     for (maskbit = 1, i = 0 ; i < 16; maskbit <<= 1)
>>     for (maskbit = 1, i = 0 ; i < 16; maskbit <<= 1)
>>     led_port[i] = '|';           /* finish with one last frame */
```

Figure 10-14 Source trace display of the subroutine outled()



```
>>     for (maskbit = 1, i = 0 ; i < 16; maskbit <<= 1)
00069 fff024a0: 7fde f214  add       r30,r30,r30
00068 fff024a4: 4bff ffb4  b         .-0x4c               ip  > FFF024A4
                                                          ip  < FFF02458
>>     for (maskbit = 1, i = 0 ; i < 16; maskbit <<= 1)
00066 fff02458: 2c1d 0010  cmpwi     r29,0x10
00065 fff0245c: 4080 004c  bge       .+0x4c
>>         led_port[i++] = '|';    /* start with a frame */
00063 fff02460: 3d80 8000  addis     r12,0,0xffff8000
00062 fff02464: 398c 0000  addi      r12,r12,0x0
00061 fff02468: 3960 007c  addi      r11,0,0x7c
00060 fff0246c: 7d6c e9ae  stbx      r11,r12,r29
00059 fff02470: 3bbd 0001  addi      r29,r29,0x1
>>         led_port[i++] = (data & maskbit)  ? ' ' : '*';
00057 fff02474: 7fea f038  and       r10,r31,r30
00056 fff02478: 554a 063e  rlwinm    r10,r10,0,24,31
00055 fff0247c: 2c0a 0000  cmpwi     r10,0x0
```

Figure 10-15 Mixed source and assembly trace display of the subroutine outled()

Example 2 – Capturing qualified trace history

Quite often there will be only a few cycles of interest out of the millions of cycles executed in a real-time run of the code. There are two ways of dealing with this:

1. Capturing, in real-time, a trace of only the cycles of interest. This is a qualified trace.
2. Post-processing (filtering out unwanted cycles) a large and expensive trace RAM buffer full of all executed and prefetched instructions.

The first is by far a quicker and more accurate method than the second. With SuperTAP's four-level event system architecture and trace control actions, you can capture a qualified trace based on a complex set of conditions, including program events, target hardware events, and the CPU bus state.

The example demonstrates this ability by capturing only the first ten memory write cycles directed to the memory-mapped LED port (address 0xA0000000).

➤ Tracing 10 writes to the LEDs

1. Restart the Cdemon program.

   In the Enter Command box of the Command window, enter:

   ```
   restart
   ```

2. Set up the qualified trace capture:

   In the Enter Command box, enter:

   ```
   trinit off
   ```

   This keeps trace capture turned off unless enabled by the event system.

3. In the Enter Command box, enter:

   ```
   trrunclr on
   ```

This clears the trace buffer when the emulator enters run mode.

4. In the Enter Command box, enter:

```
trqual dxqual
```

This configures trace for bus qualified capture.

5. In the Enter Command box, enter:

```
ctr1ival reload
```

This resets the counter to zero when entering run mode.

6. In the Enter Command box, enter:

```
when addr==dot_port && status==wr then trone,ctr1inc
```

Note: dot_port holds the base address of the LEDs (0xA0000000).

7. In the Enter Command box, enter:

```
when ctr1==10 then break
```

➤ To run the target code

- Click the Go button.

  SuperTAP breaks emulation after tracing the first 10 writes to the memory-mapped I/O address.

➤ To display trace

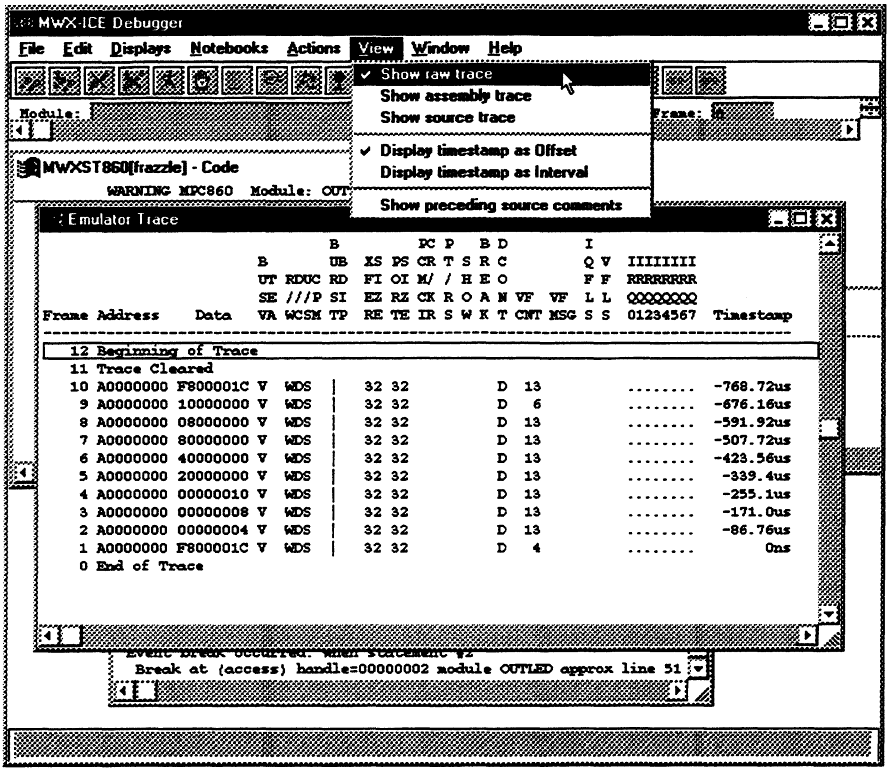
1. From the Displays menu, choose Emulator Trace.
2. From the View menu, select Show raw trace.

   You may need to scroll to see all of the trace. Only the 10 write cycles have been captured (Figure 10-16).

➤ To clear the event system

- In the Enter Command box, enter:

```
whenclr all
```



```
                                   B          PC P   B D          I
                          B        UB  XS PS  CR T S R C          Q V  IIIIIIII
                          UT RDUC  RD  FI OI  M/ / H E O          F F  RRRRRRRR
                          SE ///P  SI  EZ RZ  CK R O A N VF   VF  L L  QQQQQQQQ
Frame Address     Data    VA WCSM  TP  RE TE  IR S W K T CNT  MSG S S  01234567  Timestamp
-----------------------------------------------------------------------------------------
   12 Beginning of Trace
   11 Trace Cleared
   10 A0000000 F800001C V  WDS   |   32 32            D  13             ........  -768.72us
    9 A0000000 10000000 V  WDS   |   32 32            D   6             ........  -676.16us
    8 A0000000 08000000 V  WDS   |   32 32            D  13             ........  -591.92us
    7 A0000000 80000000 V  WDS   |   32 32            D  13             ........  -507.72us
    6 A0000000 40000000 V  WDS   |   32 32            D  13             ........  -423.56us
    5 A0000000 20000000 V  WDS   |   32 32            D  13             ........   -339.4us
    4 A0000000 00000010 V  WDS   |   32 32            D  13             ........   -255.1us
    3 A0000000 00000008 V  WDS   |   32 32            D  13             ........   -171.0us
    2 A0000000 00000004 V  WDS   |   32 32            D  13             ........  -86.76us
    1 A0000000 F800001C V  WDS   |   32 32            D   4             ........       0ns
    0 End of Trace
```

```
Break at (access) handle=00000002 module OUTLED approx line 51
```

Figure 10-16 Qualified trace capture of 10 writes to the LEDs

Example 3 – Capturing and viewing trace while running

In this example, SuperTAP emulates in dynamic mode; you can display trace while still emulating. You capture the **run()** function in trace using the event system, then view the trace in raw and disassembled formats.

➤ To restart the Cdemon program

- In the Enter Command box of the Command window, enter:

```
restart
```

➤ To set up the initial trace condition

1. In the Enter Command box, enter:

```
trinit off
```

This keeps trace capture turned off unless enabled by the event system or the **trace on** command.

2. In the Enter Command box, enter:

```
trrunclr on
```

This clears the trace buffer when the emulator enters run.

3. In the Enter Command box, enter:

```
trqual dxqual
```

This configures trace for disassembly qualified capture.

➤ To set up event system

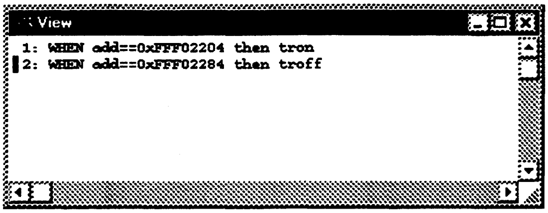
1. In the Enter Command box, enter:

```
when add==0xFFF02204 then tron
```

2. In the Enter Command box, enter:

```
when add==0xFFF02284 then troff
```

The Command window should display the two event system statements shown in Figure 10-17.



```
1: WHEN add==0xFFF02204 then tron
2: WHEN add==0xFFF02284 then troff
```

Figure 10-17 Event system statements to turn trace capture on and off

➤ To enter dynamic run mode

- In the Enter Command box, enter:

  `drun`

➤ To display a “snapshot” of trace while still running target code

1. In the Enter Command box, enter:

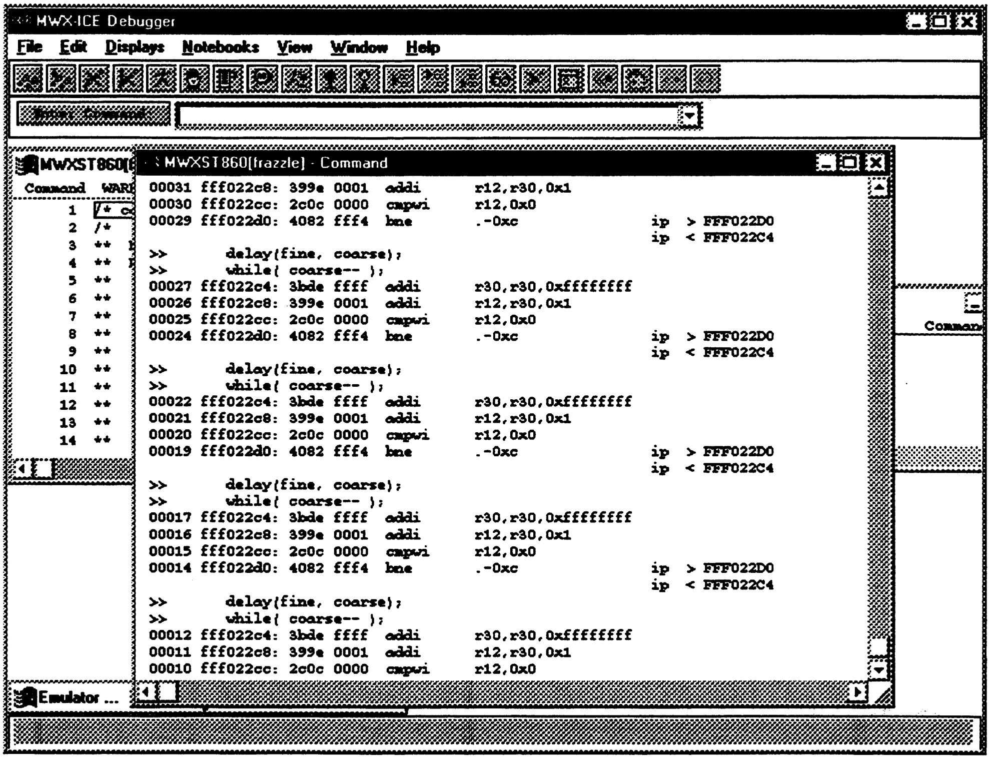
   `drt`

   This displays trace history in raw format in the Command window.

2. In the Enter Command box, enter:

   `dtb`

   This displays trace in mixed source and assembly mode (**trdisp both** is the default mode). Assembly and source-level only modes are also available using the **trdisp** command.


MWX-ICE Debugger
File Edit Displays Notebooks View Window Help
Enter Command
MWXST860(f
MWXST860[frazzle] - Command
Command WARI
Emulator ...
Command

```
00031 fff022c8: 399e 0001  addi       r12,r30,0x1
00030 fff022cc: 2c0c 0000  cmpwi      r12,0x0
00029 fff022d0: 4082 fff4  bne        .-0xc                 ip  > FFF022D0
                                                            ip  < FFF022C4
>>        delay(fine, coarse);
>>        while( coarse-- );
00027 fff022c4: 3bde ffff  addi       r30,r30,0xffffffff
00026 fff022c8: 399e 0001  addi       r12,r30,0x1
00025 fff022cc: 2c0c 0000  cmpwi      r12,0x0
00024 fff022d0: 4082 fff4  bne        .-0xc                 ip  > FFF022D0
                                                            ip  < FFF022C4
>>        delay(fine, coarse);
>>        while( coarse-- );
00022 fff022c4: 3bde ffff  addi       r30,r30,0xffffffff
00021 fff022c8: 399e 0001  addi       r12,r30,0x1
00020 fff022cc: 2c0c 0000  cmpwi      r12,0x0
00019 fff022d0: 4082 fff4  bne        .-0xc                 ip  > FFF022D0
                                                            ip  < FFF022C4
>>        delay(fine, coarse);
>>        while( coarse-- );
00017 fff022c4: 3bde ffff  addi       r30,r30,0xffffffff
00016 fff022c8: 399e 0001  addi       r12,r30,0x1
00015 fff022cc: 2c0c 0000  cmpwi      r12,0x0
00014 fff022d0: 4082 fff4  bne        .-0xc                 ip  > FFF022D0
                                                            ip  < FFF022C4
>>        delay(fine, coarse);
>>        while( coarse-- );
00012 fff022c4: 3bde ffff  addi       r30,r30,0xffffffff
00011 fff022c8: 399e 0001  addi       r12,r30,0x1
00010 fff022cc: 2c0c 0000  cmpwi      r12,0x0
```

Figure 10-18 Dynamic trace display - disassembled source and assembly

➤ To exit dynamic run mode and clear the event system

1. In the Enter Command box, enter:

   ```
   dstop
   ```

2. In the Enter Command box, enter:

   ```
   whenclr all
   ```

   This clears all event system statements.

Example 4 – Displaying data structures

Symbols include arrays, structures, static variables, register-based, and stack-based variables. Symbols can be displayed or changed by name, as declared in your program.You can display the type and scope of each symbol, and its value in binary, hex, ASCII, or decimal format.

➤ To restart the Cdemon program

- In the Enter Command box of the Command window, enter:

  ```
  restart
  ```

➤ To run to house(), a function in data()

- In the Enter Command box, enter:

  ```
  g DATA\house
  ```

  Emulation breaks at the beginning of house(), a function within data(). At this point, all of the cards have been dealt.

➤ To display the structure players

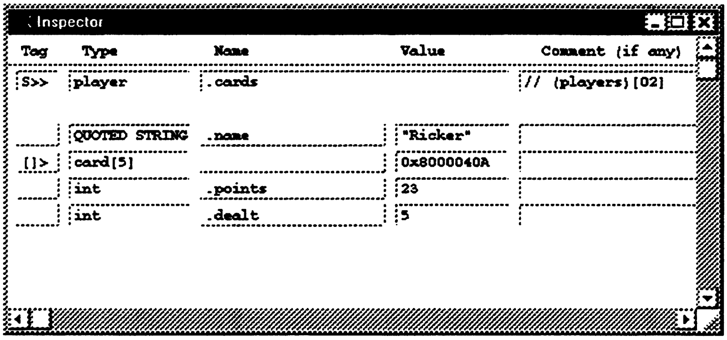
1. From the Displays menu, choose Inspector.
2. In the Inspect Symbol or Expression box, enter the following:

   ```
   players
   ```

   The Inspector window displays a break down of players. You can scroll or resize the Inspecting window for the best display of the structure.
3. From the View menu, choose Show [char*] as String.
4. In the Inspector window, choose S>> for player[02].

   The Inspector window displays a break down of player[02], the third element of players.

   In Figure 10-19, you can see player1 busted, having been dealt 5 cards for a total of 23 points. Too bad.
5. Close the Inspector window.



Tag	Type	Name	Value	Comment (if any)
S>>	player	.cards		// (players)[02]
	QUOTED STRING	.name	"Ricker"	
[]>	card[5]		0x8000040A	
	int	.points	23	
	int	.dealt	5	

Figure 10-19 Displaying the structure players

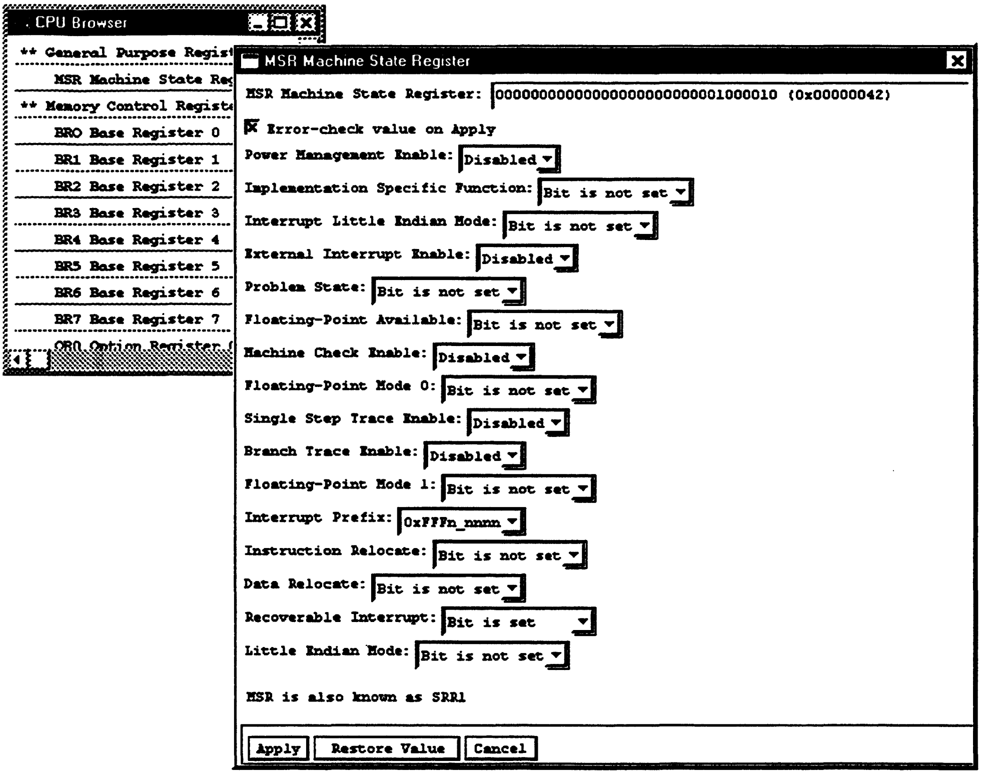
Example 5 – Browsing the CPU registers

MWX-ICE includes a register utility called **the CPU Browser,** to save you time you might otherwise spend referencing a technical manual for register bit meanings.

➤ **To browse the SR register**

1. From the Displays Menu, choose CPU Browser.
2. From the CPU Browser window, click the MSR button

   The CPU Browser displays the current MSR register settings.
3. Close the CPU Browser window.



MSR Machine State Register: 00000000000000000000000001000010 (0x00000042)

☒ Error-check value on Apply

Power Management Enable: Disabled

Implementation Specific Function: Bit is not set

Interrupt Little Endian Mode: Bit is not set

External Interrupt Enable: Disabled

Problem State: Bit is not set

Floating-Point Available: Bit is not set

Machine Check Enable: Disabled

Floating-Point Mode 0: Bit is not set

Single Step Trace Enable: Disabled

Branch Trace Enable: Disabled

Floating-Point Mode 1: Bit is not set

Interrupt Prefix: 0xFFFn_nnnn

Instruction Relocate: Bit is not set

Data Relocate: Bit is not set

Recoverable Interrupt: Bit is set

Little Endian Mode: Bit is not set

MSR is also known as SRR1

Figure 10-20 The CPU Browser view of the Status register

Example 6 – Displaying and modifying memory

MWX-ICE provides means for acting on a block of memory. Using either the command line or the Memory Commands notebook, you can clear, move, set, read, write, or log a block of memory.

➤ To restart the Cdemon program

- In the Enter Command box of the Command window, enter:

```
restart
```

➤ To display memory

- In the Enter Command box, enter:

```
dump &led_port
```

This displays a block of memory in the Command window beginning with the address of led_port. (Note: You can also use the Memory window. To open this window, choose Memory from the Displays menu.)

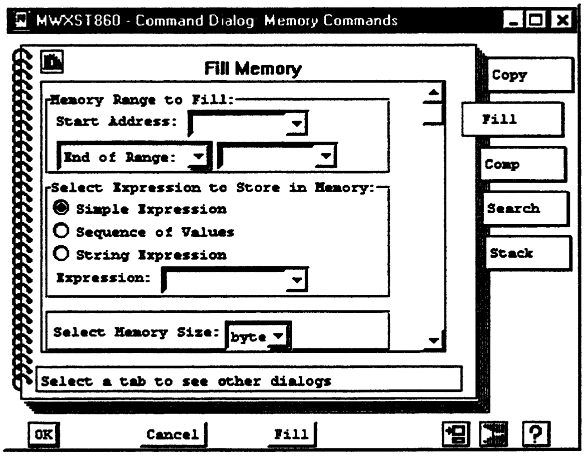
➤ To manipulate a block of memory

1. From the Notebooks menu, choose Memory Commands.
2. The Memory Commands menu, choose one of five commands.

The Memory commands notebook is displayed. You can select notebook pages by clicking the notebook tabs. The notebook pages control memory commands:

- ❑ Fill—fill memory with a given value.
- ❑ Copy—copy the contents of one block of memory to another.
- ❑ Compare—compare the contents of two memory blocks.
- ❑ Search—search through memory for a pattern.
- ❑ Stack—display values from a particular stack level.

The Fill page is shown in Figure 10-21.



MWXST860 - Command Dialog Memory Commands

Fill Memory

Memory Range to Fill:

Start Address:

End of Range:

Select Expression to Store in Memory:

● Simple Expression

○ Sequence of Values

○ String Expression

Expression:

Select Memory Size: byte

Select a tab to see other dialogs

Figure 10-21 Memory Commands notebook - Fill memory page

Example 7 – Monitoring and modifying variables

MWX-ICE provides three windows for working with program variables. The Data window is used for monitoring variables, the Inspector window for viewing and modifying variables, and the Register window for viewing and modifying register-based variables. In static mode, they are updated only at each single-step, breakpoint, or program halt. In dynamic mode, emulation periodically pauses then re-starts, updating each window when emulation is re-entered.

➤ To dynamically monitor the variable led_port

1. In the Enter Command box of the Command window, enter:

   ```
   g outled
   ```

2. From a Displays menu, choose Data.
3. In the Data window Expression box, type **led_port**, and then click the Display button:

   This places the variable led_port in the Data window.

➤ To restart the Cdemon program

- In the Enter Command box of the Command window, enter:

  ```
  restart
  ```

➤ To enter dynamic run mode

- In the Enter Command box, enter:

  ```
  drun
  ```

➤ To enter dynamic update mode

- In the Enter Command box, enter:

  ```
  dupdate 1000
  ```

  This causes MWX-ICE to poll the emulator approximately 1000 times a minute. The response is actually slower due to the time spent updating the debugger's windows.

Observe the variable led_port in the Data window as it changes.

➤ To exit dynamic update mode

- Choose Stop from the tool bar.

➤ To dynamically modify the variable direct

1. From the Displays menu, choose Inspector.
2. In the Inspector window Symbol Name box, enter the following:

   ```
   direct
   ```

   The variable direct controls the direction of the LED's counting, either left or right.
3. In the Inspector window, choose the long button to the left of the value left.

   This opens up a dialog box used to change the value of the inspected variable, shown in Figure 10-23.
4. In the Enter new value field of the dialog box, type:

   ```
   right
   ```
5. In the dialog box, choose Set to accept the new value.

   The Inspector window for direct now reflects the new value.
6. Enter dynamic update mode again:

   ```
   dupdate 1000
   ```

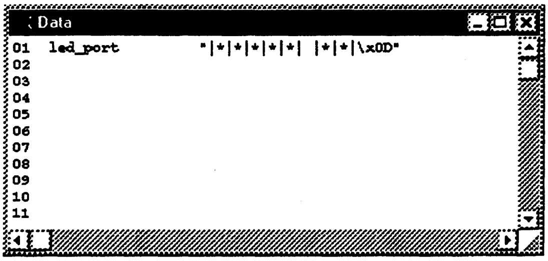
   Observe the motion of the asterisks displayed in the Data window.
7. Choose Stop from the tool bar to exit dynamic update mode.

➤ To exit dynamic run mode and return to pause mode

- In the Enter Command box, enter:

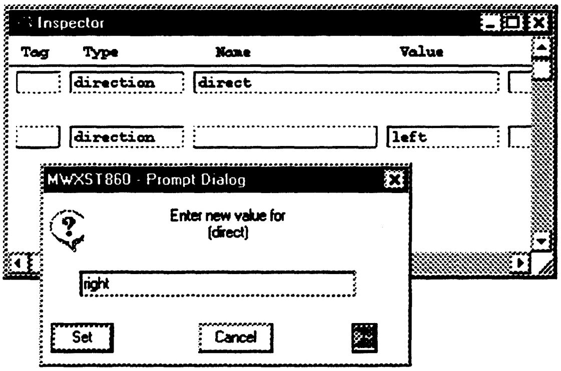
  ```
  dstop
  ```

  This forces SuperTAP from dynamic run mode back into pause mode.



```
01  led_port          "|*|*|*|*|*| |*|*|\x0D"
02
03
04
05
06
07
08
09
10
11
```

Figure 10-22 Dynamic mode - Monitoring the variable led_port



Tag	Type	Name	Value
	direction	direct	
	direction		left

Enter new value for
(direct)

right

Figure 10-23 Dynamic mode - Entering a variable's new value

Appendix A
Modifying the Startup Files

To save you time in setup and configuration of MWX-ICE, you can save your emulator connection and configuration information, and your startup command-line arguments. MWX-ICE saves this information in two separate files. This appendix describes how these two startup files interact, and describes the process used by MWX-ICE at startup.

Contents **Page**

How MWX-ICE uses the startup files

When you start MWX-ICE, the debugger automatically looks for the MWX.CFG file to obtain the *command-line* or startup options. The MWX.CFG file is created by the Startup Options Editor. When you first use MWX-ICE, you need to run the Startup Options Editor to select the correct processor type. Using the Editor, you can also specify an absolute file to download, select journal or log files to record your debugging session, include command files. You can also specify a Startup Settings file to use, which then overrides the default file (STARTUP.INC).

After MWX-ICE has obtained the command-line arguments, it looks for the STARTUP.INC file, or for a Startup Settings file if you specified one using the Startup Options Editor. This file contains the emulator configuration and connection commands, and is processed before any other include files. MWX-ICE automatically looks for this file at startup. The STARTUP.INC file is created when you save your configuration.

After all the commands in the STARTUP.INC file have been executed, MWX-ICE finishes processing the rest of the command-line options.

For an illustration of how MWX-ICE uses the startup files, see Figure A-1.

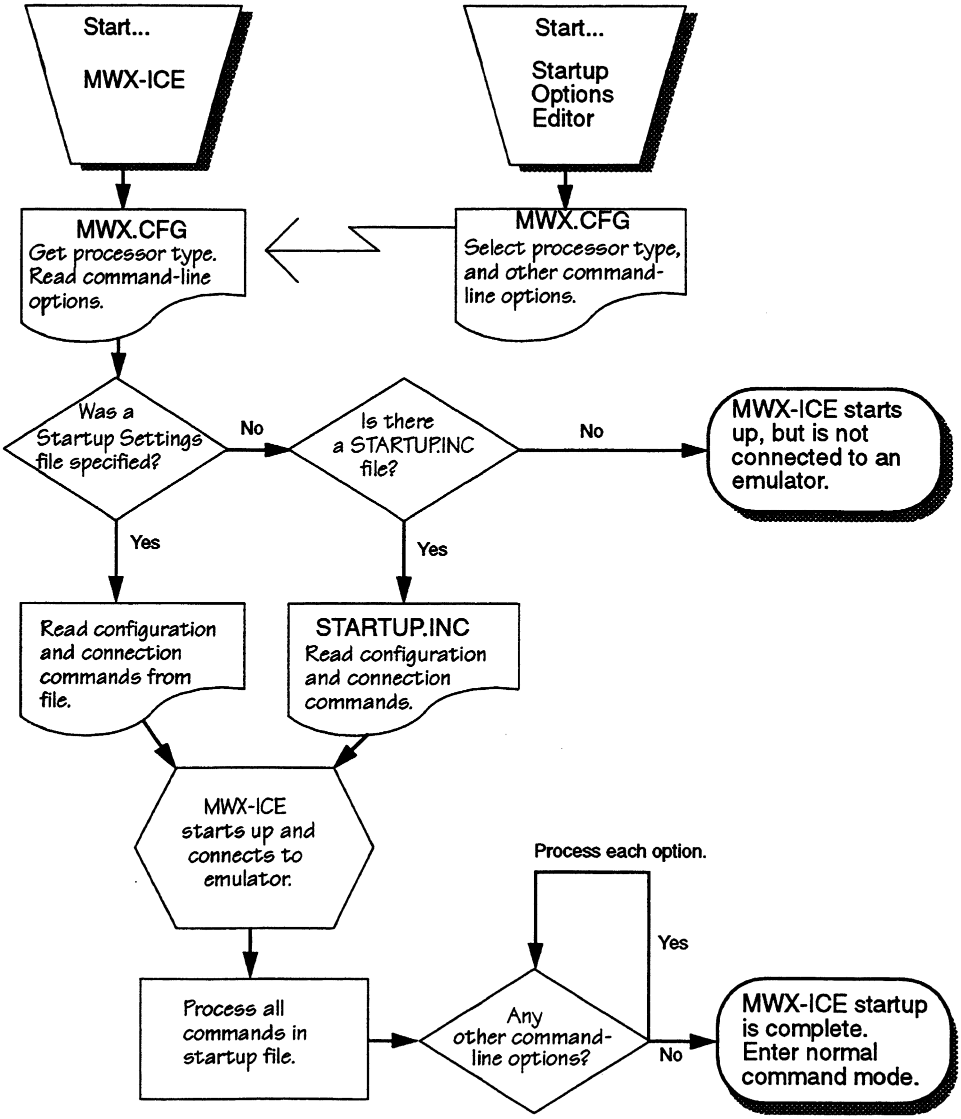
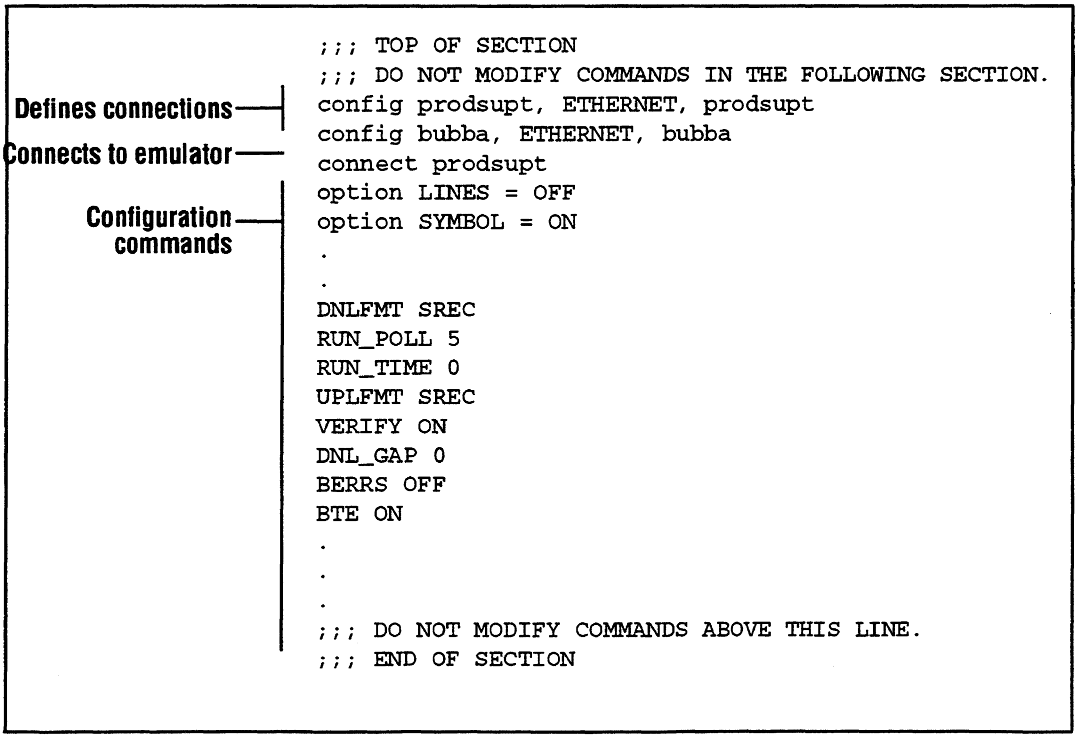

Start...
MWX-ICE
Start...
Startup Options Editor
MWX.CFG
Get processor type. Read command-line options.
MWX.CFG
Select processor type, and other command-line options.
Was a Startup Settings file specified?
No
Is there a STARTUP.INC file?
No
MWX-ICE starts up, but is not connected to an emulator.
Yes
Yes
Read configuration and connection commands from file.
STARTUP.INC
Read configuration and connection commands.
MWX-ICE starts up and connects to emulator.
Process each option.
Yes
Process all commands in startup file.
Any other command-line options?
No
MWX-ICE startup is complete. Enter normal command mode.

Figure A-1 How MWX-ICE uses the STARTUP.INC and MWX.CFG files.

About the startup file (STARTUP.INC)

The startup file is a special kind of include file that is processed before any other include files. This file has a configuration section that is composed of three parts: the first part contains the definitions of emulator connections (using the **config** command); the second part connects to a specific emulator (using the **connect** command); the third part contains the configuration commands. Do not edit the configuration section of the startup file. The configuration commands cannot be processed if MWX-ICE is not connected to an emulator.


Defines connections
Connects to emulator
Configuration commands

```
;;; TOP OF SECTION
;;; DO NOT MODIFY COMMANDS IN THE FOLLOWING SECTION.
config prodsupt, ETHERNET, prodsupt
config bubba, ETHERNET, bubba
connect prodsupt
option LINES = OFF
option SYMBOL = ON
.
.
DNLFMT SREC
RUN_POLL 5
RUN_TIME 0
UPLFMT SREC
VERIFY ON
DNL_GAP 0
BERRS OFF
BTE ON
.
.
.
;;; DO NOT MODIFY COMMANDS ABOVE THIS LINE.
;;; END OF SECTION
```

Figure A-2 The configuration section of an example of a STARTUP.INC file

Where MWX-ICE looks for the STARTUP.INC file

MWX-ICE searches the following directories for the STARTUP.INC file: (note that XRAYMASTER is the directory pointed to by the XRAYMASTER environment variable):

- ❑ *current_directory*
- ❑ *current_directory*\AMC\ST8XX
- ❑ XRAYMASTER\AMC\ST8XX
- ❑ C:\ST8XX\AMC\ST8XX

XRAYMASTER is the directory pointed to by the XRAYMASTER environment variable. If the STARTUP.INC file isn't found, MWX-ICE comes up in an unconnected state. By default, when you save a configuration to the startup file, you are prompted to save the connection and the configurations to the STARTUP.INC file in your C:\ST8XX\AMC\ST8XX directory.

Adding commands to the startup file

You can add commands not normally saved as part of configuration. Using a text editor, place such commands after the configuration section of the startup file. This section ends with:

```
;;; DO NOT MODIFY COMMANDS ABOVE THIS LINE.
;;; END OF SECTION
```

For example, you could call an additional include file:

```
;;; DO NOT MODIFY COMMANDS ABOVE THIS LINE.
;;; END OF SECTION
inc myregs.inc
```

To preserve such additions to the file, always use Replace, rather than Overwrite, when saving new configuration data to the file.

To process the commands at startup, MWX-ICE must be connected to an emulator. For a list of all the commands and their descriptions, see MWX-ICE Help.

Including a specific startup file

By default, MWX-ICE searches for the STARTUP.INC file. You can override an existing startup file by specifying another file to use.

➤ **To include a specific startup file**

- From the Startup Options Editor, select a Startup Settings file.

Changing connections during a debug session

Once MWX-ICE is running, you can run a startup include file as long as you are not connected to an emulator. Just make sure you are connecting to an emulator that matches the processor type you selected using the Startup Options Editor.

➤ **To include a startup file from MWX-ICE**

- From the Connections window File menu, choose Run Include. This opens the Include page of the Debugger Files notebook. Type the name of the file, or use the File Chooser, and then click Include.

—or—

- Use the **include** command from MWX-ICE Command window.

Restoring emulator default settings

Once you create a startup include file, MWX-ICE uses the definitions and configuration in that file. If you want to reset the emulator to its default state, follow these steps.

➤ **To restore default settings**

1. Exit MWX-ICE.
2. Turn off power to the target and emulator.
3. Rename or delete the existing STARTUP.INC file(s).

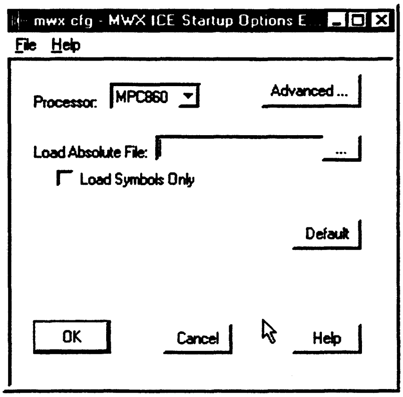
Note that MWX-ICE looks for the STARTUP.INC file in several locations. See "Where MWX-ICE looks for the STARTUP.INC file" on page A-5.

4. Start MWX-ICE.
5. Turn on power to the emulator and target.
6. Define a connection, and connect to an emulator.

About the startup options file (MWX.CFG)

To make selecting startup options easier, you can use the MWX-ICE Startup Options Editor. The editor creates a startup options file (MWX.CFG) that is automatically included when you start the debugger.

The debugger automatically looks for the file MWX.CFG, and uses those options at startup. If you save the options to a different file, you need to specify the name of the file in the command line at startup.


mwx cfg - MWX ICE Startup Options E...

Processor: MPC860

Load Absolute File:

□ Load Symbols Only

Figure A-3 MWX-ICE Startup Options Editor

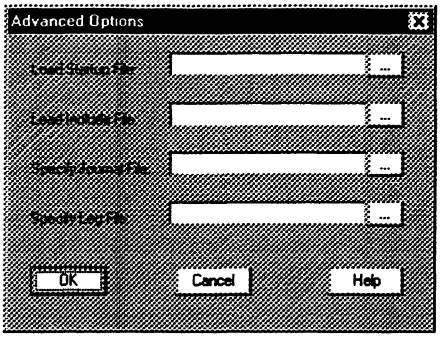


Figure A-4 MWX-ICE Startup Options Editor Advanced Options

Specifying which startup option file to use

The debugger automatically looks for the file MWX.CFG, and uses those options at startup.

If you save the options to a different file, you need to add a new program icon to the Start or Programs menu. For information, click Help from the Startup Options Editor, and see the "Add a new program" topic.

Selecting startup options

You can select the following startup options. For more information about each option, choose Help from the Help menu, click the Help button, or press F1.

Table A-1 Basic options provided by the Startup Options Editor

Options	Definition
Processor	When more than one, selects the processor variant that you want. This is the only change necessary to switch variants.
Load Symbols Only	Loads only the symbols of the file specified in the Load Absolute File field.
Load Absolute File	The name of an absolute object module that is loaded into the debugger's simulated target memory. The default extension is .ABS. Use an absolute pathname, if the file is not in the directory from which MWX-ICE was started; or set the XRAY environment variable to point to source locations. Only one absolute object module may be loaded into MWX-ICE via the startup editor. To load multiple absolute object modules, use the **load** command after starting the debugger.

Table A-2 Advanced options provided by the Startup Options Editor

Options	Definition
Load Include File	The name of an included command file that is read before any debugger commands are entered. The default file extension is .INC. Use an absolute pathname, if the file is not in the directory from which MWX-ICE was started; for example, \PROJECT\JOHN\SWITCHES.INC.

Table A-2 Advanced options provided by the Startup Options Editor

Options	Definition
Specify Journal File	Command output and window information is saved in the specified *journal_file*. The default extension is .JOU. This file is placed in the directory from which MWX-ICE was started, unless you specify a complete path; for example, \PROJECT\ADMIN\TEST5.JOU.
Specify Log File	User commands and a record of any errors are placed in the specified *log_file*. The default extension is .LOG. This file is placed in the directory from which MWX-ICE was started, unless you specify a complete path; for example, \PROJECT\ADMIN\TEST5.LOG.

Appendix B
Troubleshooting

The following list covers the most common problems that occur during installation and using MWX-ICE. They are grouped according to when the failure typically occurs:

- ❑ During startup.
- ❑ During normal operation.

Each group consists of one or more problems. Locate the problem that suits your situation, and check the causes and solutions listed. Where problems are specific to a particular host or communications type, it is noted.

Common startup problems

Here are some common problems that you might encounter when you try to run the MWX-ICE debugger.

If MWX-ICE fails to start, or exits suddenly, check the following possible causes.

Insufficient memory

To use MWX-ICE you must have a 386-class personal computer (or better) running MS Windows 95 or Windows NT. You need at least 16 MB of RAM and 20 MB of free swap space (Virtual Memory).

An MWX-ICE support file was not found

An MWX-ICE support file is missing or not in the search path. MWX-ICE searches the directories in the following order:

- ❑ *current_directory*\STARTUP.INC
- ❑ *current_directory*\AMC\ST8XX\
- ❑ XRAYMASTER[1]\AMC\ST8XX\
- ❑ C:\ST8XX\AMC\ST8XX\

The default installation directory for MWX-ICE is C:\ST8XX. If you use this directory, you do not need to set up the XRAYMASTER environment variable.

If you have installed the MWX-ICE debugger in different directory, you need to set up the XRAYMASTER environment variable in your AUTOEXEC.BAT file. The syntax is:

```
set XRAYMASTER=install_dir
```

MWX-ICE reports it cannot find the necessary shell file

The emulator control file is missing or not in the search path.

[1] Directory pointed to by the XRAYMASTER environment variable.

The search order that MWX-ICE uses is shown on page B-2

Check these locations for the .ep file. If you are not using the default installation directory (C:\ST8XX), be sure to set the XRAYMASTER environment variable to point to your installation directory.

Error while opening a connection to the emulator

MWX-ICE reports that the debug server is initializing, but nothing happens.

Check the following:

- □ Is the emulator is connected to a power and is the power switch is on?
- □ Is the emulator connected properly to the host computer or network?
- □ Is STARTUP.INC file set up correctly for the type of communications you are using?
- □ Is the emulator properly installed on your network?

Check the communications type

Check that the Connection Type for the defined connection matches the switch settings on the emulator. For information about communications setup, see “Defining a connection” on page 2-9.

Check emulator connections

If you are using Ethernet communications, tighten the Ethernet cable connection between the emulator and network.

Check the host name for the emulator

The host name you are using for the emulator and the host name specified in the Connections window must match.

If your TCP/IP application provides the PING program, you can use it to test the connection to the emulator. PING sends ICMP_ECHO_REQUEST packets to a remote host. From your computer you can *ping* the emulator to determine if it is accessible. The PING command should report the emulator to be alive. If not, the emulator has been installed incorrectly, and you might need the assistance of your network administrator to correct it. Verify each step of the Ethernet installation given in the hardware manual. Check your network HOSTS files or domain name system services, and your TCP/IP setup.

If the PING works, shut the emulator off and try again. You want to make sure that you are indeed talking to the emulator. With the emulator off, the ping should fail. If this happens, you have the correct address of the emulator, so turn it back on.

Check WINSOCK.DLL

The file WINSOCK.DLL must be Windows Sockets version 1. 2 compliant. Make sure this file is in your Windows directory, or in your environment path. Without a version 1.2 compliant WINSOCK.DLL, you won't be able to connect to the emulator.

If you get the error message "Error while opening a connection to the emulator," and you have the PING utility, try to verify that the emulator is on the network. If the emulator is on the network and you've used the correct host name for the emulator, you might need a newer version of WINSOCK.DLL.

Does your network use IEEE 802.3 frames?

The emulator communicates with the host using the standard IEEE 802.3 packets. Be sure your network supports the IEEE 802.3 format.

Download errors

The following section lists some common errors that may occur during downloads to the target or to the emulator's overlay memory.

Virtual memory simulator failure

MWX-ICE swaps symbols to disk and needs more disk space to continue. Clear disk space and try again.

Lack of symbol space

Not enough extended memory.

MWX-ICE is storing symbols in extended memory, and has run out of space. You need at least as much memory available as the size of the object file being loaded, and usually you need about 25% more. So, if you are loading a 4 MB object file, you will need at least 16 MB of memory in your PC (12 for MWX-ICE, and 4 for the symbolic information).

Miscellaneous errors

MWX-ICE reports the shell is newer or older than expected

When you first run a new version of MWX-ICE, it is a good idea to select the Force OS Download in the Connections window, so the operating system in the emulator can be updated with the latest version provided on the distribution.

Note that after you cycle power to the emulator, the emulator OS is automatically downloaded when you start the MWX-ICE debugger.

Check paths and the XRAYMASTER variable

You are encouraged to keep only the most current version of MWX-ICE on your system. If you choose to keep earlier versions available, be sure to modify all path statements, environment variables, and user files containing file pointers (includes, start-up file etc.) to search the appropriate new paths.

The search order that MWX-ICE uses is shown on page B-2.

Check these locations for the .EP file. Remove any obsolete .EP from those locations, or reset your XRAYMASTER environment variable.

Corrupted file

If you suspect the .EP file is corrupted for some reason, try reinstalling MWX-ICE. After you install the debugger, select the Force OS Download in the Connections window, so the operating system in the emulator can be updated when you start the debugger.

If MWX-ICE still reports a version error, please call Customer Support at 1-800-ASK-4-AMC.

MWX-ICE reports "type errors" or "mismatched variables"

A function or variable has been declared as more than one type in different modules. Symbols for a module load only when that module comes in scope. When one of these warnings occurs, use **ps /e/f** to print the mismatch. You might need to use the **scope** *module\symbol* command in conjunction with the **ps** command to see the additional declarations of the symbol.

If you want to see all mismatches at startup, set **option demandload=off**; then load your code. A **ps /e/f** command prints all known mismatches. Note that with **demandload off**, downloads can take considerable time, and you might run out of symbol space.

Calling Customer Support

If none of the suggested solutions works, please contact the Applied Microsystems Customer Support department (US and Canada) or your nearest Applied Microsystems sales office (overseas).

When you contact Customer Support, please have the following information available:

- ❑ The ASI number of your system. The ASI number is printed on a label located on the bottom of the SuperTAP.
- ❑ Your Support Agreement number (if applicable).
- ❑ The emulation device you are connected to (68040 emulator, 68332 CodeTAP, etc.).
- ❑ The available conventional and extended RAM in your PC.
- ❑ The version of MWX-ICE you are using, as well as the information reported by the **hwconfig** command in the debugger (assuming you can get this).
- ❑ The exact sequence of operations and commands that immediately preceded your problem.

Phone

(800) ASK-4AMC (275-4262)

(206) 882-2000 (in Washington or from Canada)

See inside back page for addresses and phone numbers of worldwide offices.

Internet

If you have access to the Internet, you can contact Applied Microsystems Customer Support using the following address:

support@amc.com

If you have access to the World Wide Web, check out the Applied Microsystems home page:

http://www.amc.com

FAX

If you prefer, you can fax your problem description to us. Be sure to include the information requested above.

(206) 883-3049

Appendix C
Cdemon Demonstration Program

Cdemon is the Applied Microsystems standard C-language demonstration program, providing examples of many code and data constructions used by C programmers. An in-memory representation of the LEDs (*led_port*) may be used to see the output of some of the functions.

Cdemon is composed of two discrete demonstration programs. The default C-language program writes to the LEDs and plays a simple hand of blackjack. The other, a C++-language program, simulates an elevator. A variable named *which_demo* determines which of the two demonstration programs is executed. The card game program runs by default. If *which_demo* is set to 1, then the elevator program is executed.

A functional block of the LED-lighting/blackjack game is shown in Figure C-1.

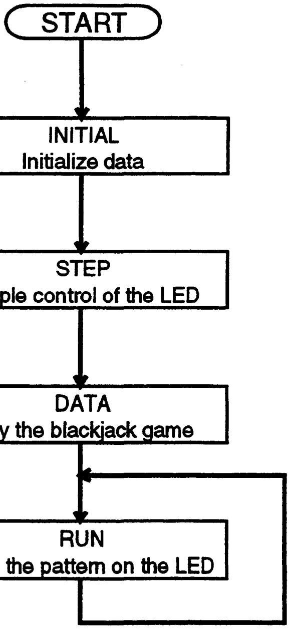


Figure C-1 Flowchart of Cdemon

initial()

The function initial() initializes the three global LED-control variables, *pattern*, *speed*, and *direct*. These variables control the byte pattern, the speed, and the direction of LED pattern rotation, respectively.

After initializing the global variables, initial() passes control to step().

step()

The function step() performs five loops of a simple LED control process, and passes control to data().

Step() declares the loop-control variable *loops*. In each of its five loops, step() calls outled() 17 times, each time passing outled() a one-byte argument which represents the pattern displayed on the demonstration board LEDs.

Outled() then writes the LED control byte to the LEDs and to an in-memory representation of the LEDs (symbolically named *led_port*). Each of the 17 arguments passed to outled() by each loop in step() represents one LED pattern.

While the execution of step() can be observed with the trace function, the purpose of step() is to demonstrate, in a single-stepped fashion, the relationship between the code and the LEDs.

The most basic control of step() comes from single-stepping while observing and modifying the loop-control variable *loops*. *Loops* may be observed with the Data window, and observed or changed with the Inspector window. Setting *loops* to a high value will lengthen the time spent in step(), while setting *loops* to zero will very quickly cause program control to be passed to data().

Because step() produces a repeating cycle of data on the bus, predictable data-value conditions are available to the event system.

data()

The function data() plays a five-handed game of blackjack with four players and a dealer. When the game is won, data() passes control to run(). Data() requires no input and generates no output, and is simply a code environment with interesting data structures.

The primary data structures are:

1. *card*, a structure defining the value and suite of each card.
2. *player*, a structure containing each player's name, a hand (array) of five cards, a point total, and a card count.

3. *card_deck*, a union with various array types describing the cards in the four suits, the cards in the two sub-decks for shuffling, and the cards in the shuffled deck.

After the declarations, data() initializes player names and sets cards dealt and points for each player to zero, then executes the following functions.

init()

This function initializes players and dealer structures.

sort()

This function sets up a 52-word block of memory as a deck of cards.

shuffle()

This function divides the deck into two 26-word blocks and interleaves them, simulating the shuffling process.

deal()

This function deals one card to each player, including the dealer.

hit()

This function deals cards to each player until *points* >= 18, or cards dealt = 5.

house()

This function deals cards to the dealer until *points* >= 17, or cards dealt = 5.

The players each draw for cards while they have less than 21 points and more than 18 points. The dealer uses a similar routine to draw cards until his hand contains more than 17 points. The game concludes after one round.

The data() function only executes once. To replay the game, reset the program to return to the beginning of the code, and run the code until it reaches a temporary breakpoint set at the beginning of the data() function.

run()

The function run() writes a string from left to right (or right to left, depending on the value of the variable *direct*) to the LED's endlessly. Rather than using separate statements like step(), run() uses a "while" control structure under the direction of the *speed* and *direct* variables from initial(). Program control stays with run().

Run() declares external functions outled() and wait(), declares the byte *maskbit*, the integer *cputype*, the loop-control variable *i*, and the constant *forever = 1*.

outled()

This function writes an 8-bit value to *led_port*, the in memory representation of LEDs.

wait()

This function sets the actual delay according to the value of two arguments, *cputype* and *speed*.

The mechanics of run() can be observed by changing the values of *direct* and *speed*, and then running without breakpoints. The effects of changed variables in the LED-control task can be observed directly at the LEDs or at the Data window while monitoring *led_port*.

Appendix D
Updating the SuperTAP Flash ROM

At some point you may be required to update the emulator firmware stored in the SuperTAP's flash memory. Typically this occurs when you are installing an update to existing software, and the release letter specifies a required higher level of emulator core software.

This appendix describes the steps in the upgrade process.

Part 1: Determine the current core version (optional)

To determine the current level of the emulator core software, you can use an MWX-ICE command.

➤ To check the core version

1. Start MWX-ICE.

   Note that the most recently installed version of MWX-ICE may be incompatible with the core currently on the SuperTAP and fail to start. If this occurs, either start an earlier version of MWX-ICE, or skip this part and go directly to Part 2.

2. Connect to the SuperTAP.
3. Enter **hwconfig** on the command line.
4. Compare the Core System and Core Loader versions displayed with those required by the latest release of MWX-ICE. If an upgrade is required, proceed with the remainder of the procedures in this appendix.

Part 2: Activate the SuperTAP's core loader

➤ To activate the loader

1. Turn off power to the SuperTAP.
2. Using the RJ-11 adapter provided (P/N 210-12502 or P/N 210-12503), connect the serial cable (600-12511) to the RS-232 port on your PC to the CONFIG port on the SuperTAP.
3. Start HyperTerminal or another terminal emulation program and connect to the COM port that is directly connected to the emulator.

   Specify the following configuration parameters:

   - 9600 baud

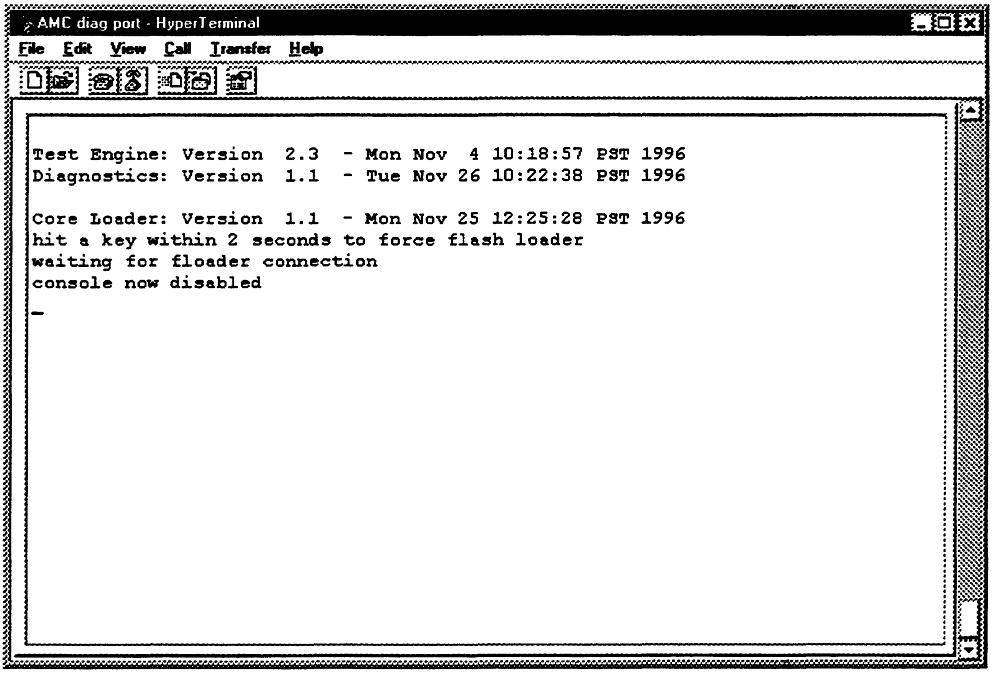
- 8 data bits
- no parity
- one stop bit
- hardware flow control

4. Apply power to the SuperTAP, and then watch the terminal screen.
5. As the SuperTAP starts up, it displays information about the current loader and test engine, as soon as you see the message, "hit a key within 2 seconds to force flash loader," quickly press a key.
6. When you successfully activate the flash loader, you should see the following message in the terminal window:

   ```
   waiting for floader connection
   console now disabled
   ```

   If you didn't press a key quickly enough, cycle power to the emulator, and try it again.
7. Once you see the message that the emulator is waiting for the floader connection, exit the terminal emulation program and disconnect from the emulator.

   When the core loader is activated both LEDs on the SuperTAP flash rapidly. Be sure to leave the SuperTAP power on.


AMC diag port - HyperTerminal
File Edit View Call Transfer Help

```
Test Engine: Version  2.3  - Mon Nov  4 10:18:57 PST 1996
Diagnostics: Version  1.1  - Tue Nov 26 10:22:38 PST 1996

Core Loader: Version  1.1  - Mon Nov 25 12:25:28 PST 1996
hit a key within 2 seconds to force flash loader
waiting for floader connection
console now disabled

_
```

Figure D-1 HyperTerminal (core loader activated)

Part 3: Program the emulator's firmware

Once you've activated the core loader, you can run the FLOADER utility that is included with MWX-ICE.

➤ **To run the FLOADER utility**

1. Using Windows Explorer, change directories to:

   `install_drive:\install_dir\UTILS`

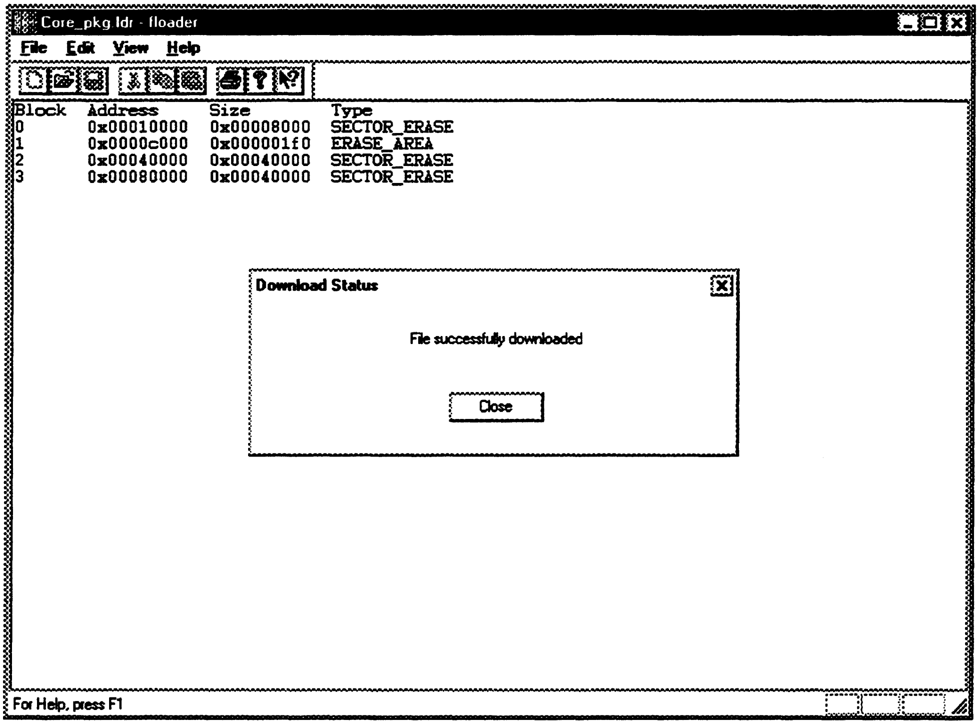
   The default is C:\ST8XX\UTILS.
2. Start the FLOADER.EXE utility.
3. Choose Settings from the Edit menu.
4. In the Communications Settings dialog box, select the communications port the emulator is connected to, and select the fastest baud rate that your PC can handle, and then click OK.
5. Choose Open from the File menu to open the flash file: CORE_PKG.LDR

   Information about the loader file is displayed in the main window.
6. From the File menu, choose Download.

   The SuperTAP automatically configures itself to the baud rate selected. This should only take a moment or two. If the autobaud is unsuccessful, try using a slower baud rate.

   FLOADER reports on the progress of the download. When the process is complete, you should see the message: File Successfully downloaded.
7. Press the reset button.

The upgrade is complete. You can exit FLOADER.EXE and reconnect the Ethernet cable to the SuperTAP.



```
Block   Address     Size        Type
0       0x00010000  0x00008000  SECTOR_ERASE
1       0x0000c000  0x000001f0  ERASE_AREA
2       0x00040000  0x00040000  SECTOR_ERASE
3       0x00080000  0x00040000  SECTOR_ERASE
```

Download Status

File successfully downloaded

Figure D-2 FLOADER utility (firmware successfully updated)

These procedures must be performed on each emulator that you plan to use with the current version of MWX-ICE.

Part 4: Remove outdated shells and pointers

Before you start MWX-ICE

Before you start MWX-ICE for the first time after reprogramming the emulator's firmware, be sure you remove any old shell files (*.EP) or EMULCFG.DAT files that point to earlier shell files. To load the latest shell file when you connect to the emulator, select the Force OS Download option in the Connections dialog box.

What's next?

Once you have completed the upgrade, return to the MWX-ICE startup procedures in Chapter 2.

Index

The Master Index contains the indexes for the three manuals that make up the documentation for the SuperTAP emulator and MWX-ICE debugger. The document set to which the manual belongs appears in parentheses following the page number reference. The following abbreviations are used:

Abbr.		**Manual Title**
(RXRY)	XRAY Debugger for Windows	*XRAY Reference Manual*
(UXRY)	XRAY Debugger for Windows	*XRAY User's Guide*
(XUM)		*MWX-ICE User's Manual*

SuperTAP MPC8XXX System Integration Tool

Abbr.		Manual Title
(RXRY)	XRAY Debugger for Windows	*XRAY Reference Manual*
(UXRY)	XRAY Debugger for Windows	*XRAY User's Guide*
(XUM)		*MWX-ICE User's Manual*

Abbr.		Manual Title
(RXRY)	XRAY Debugger for Windows	*XRAY Reference Manual*
(UXRY)	XRAY Debugger for Windows	*XRAY User's Guide*
(XUM)		*MWX-ICE User's Manual*

Abbr.		**Manual Title**
(RXRY)	XRAY Debugger for Windows	*XRAY Reference Manual*
(UXRY)	XRAY Debugger for Windows	*XRAY User's Guide*
(XUM)		*MWX-ICE User's Manual*

Abbr.		**Manual Title**
(RXRY)	XRAY Debugger for Windows	*XRAY Reference Manual*
(UXRY)	XRAY Debugger for Windows	*XRAY User's Guide*
(XUM)		*MWX-ICE User's Manual*

Abbr.		**Manual Title**
(RXRY)	XRAY Debugger for Windows	*XRAY Reference Manual*
(UXRY)	XRAY Debugger for Windows	*XRAY User's Guide*
(XUM)		*MWX-ICE User's Manual*

Abbr.		Manual Title
(RXRY)	XRAY Debugger for Windows	*XRAY Reference Manual*
(UXRY)	XRAY Debugger for Windows	*XRAY User's Guide*
(XUM)		*MWX-ICE User's Manual*

Abbr.		**Manual Title**
(RXRY)	XRAY Debugger for Windows	*XRAY Reference Manual*
(UXRY)	XRAY Debugger for Windows	*XRAY User's Guide*
(XUM)		*MWX-ICE User's Manual*

Abbr.		**Manual Title**
(RXRY)	XRAY Debugger for Windows	*XRAY Reference Manual*
(UXRY)	XRAY Debugger for Windows	*XRAY User's Guide*
(XUM)		*MWX-ICE User's Manual*

Abbr.		Manual Title
(RXRY)	XRAY Debugger for Windows	*XRAY Reference Manual*
(UXRY)	XRAY Debugger for Windows	*XRAY User's Guide*
(XUM)		*MWX-ICE User's Manual*

Abbr.		Manual Title
(RXRY)	XRAY Debugger for Windows	*XRAY Reference Manual*
(UXRY)	XRAY Debugger for Windows	*XRAY User's Guide*
(XUM)		*MWX-ICE User's Manual*

Abbr.		**Manual Title**
(RXRY)	XRAY Debugger for Windows	*XRAY Reference Manual*
(UXRY)	XRAY Debugger for Windows	*XRAY User's Guide*
(XUM)		*MWX-ICE User's Manual*

Abbr.		**Manual Title**
(RXRY)	XRAY Debugger for Windows	*XRAY Reference Manual*
(UXRY)	XRAY Debugger for Windows	*XRAY User's Guide*
(XUM)		*MWX-ICE User's Manual*

Abbr.		Manual Title
(RXRY)	XRAY Debugger for Windows	*XRAY Reference Manual*
(UXRY)	XRAY Debugger for Windows	*XRAY User's Guide*
(XUM)		*MWX-ICE User's Manual*

Abbr.		Manual Title
(RXRY)	XRAY Debugger for Windows	*XRAY Reference Manual*
(UXRY)	XRAY Debugger for Windows	*XRAY User's Guide*
(XUM)		*MWX-ICE User's Manual*

Abbr.		**Manual Title**
(RXRY)	XRAY Debugger for Windows	*XRAY Reference Manual*
(UXRY)	XRAY Debugger for Windows	*XRAY User's Guide*
(XUM)		*MWX-ICE User's Manual*

Applied Microsystems Corporation

Applied Microsystems Corporation maintains a worldwide network of direct offices committed to quality service and support. For information on products, pricing, or delivery, please call the nearest office listed below. In the United States, for the number of the nearest local office, call 1-800-426-3925.

CORPORATE OFFICE
Applied Microsystems Corporation
5020 148th Avenue Northeast
P.O. Box 97002
Redmond, WA 98073-9702
(206) 882-2000
1-800-426-3925
Customer Support
1-800-ASK-4AMC (1-800-275-4262)
TRT TELEX 185196
FAX (206) 883-3049
Internet Home Page: http://www.amc.com

EUROPE
Applied Microsystems Corporation Ltd.
AMC House
South Street
Wendover
Buckinghamshire
HP22 6EF United Kingdom
44 (0) 296-625462
Telex 265871 REF WOT 004
FAX 44 (0) 296-623460

JAPAN
Applied Microsystems Japan, Ltd.
Arco Tower 13 F
1-8-1 Shimomeguro
Meguro-ku
Tokyo 153, Japan
81-3-3493-0770
FAX 81-3-3493-7270

Part No.	Revision History	Date
924-00101-00	Initial release of the MWX-ICE Debugger for Windows (SuperTAP MPC8XX).	2/97

P/N 924-00101-00
February 1997

Applied Microsystems Corporation