Mentor® Embedded Sourcery™ Probe User's Manual

Version 2.5.0

February 2015

Document Revision 1

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form

Revision History

Revision	Changes	Date
0	Initial release.	Sept 2014
1	Added missing Mentor trademark symbols.	Feb 2015

Revision History

Table of Contents

Chapter 1 Mentor Embedded Sourcery Probe Introduction

Mentor Embedded Sourcery Probe (Sourcery Probe) is a debugging and development tool that provides you with the ability to see what is taking place in the target system, and control its behavior. This probe provides the debug services the debugger uses to perform debug operations. It receives command packets over the communication link and translates them into the Joint Test Action Group (JTAG) operations required to provide the specific services. It contains the hardware and software required to control the processor's Debug Support Unit (DSU) through the JTAG interface. It performs debug services such as memory test, single stepping, and breakpoint management in addition to the following tasks:

- Resets the target system
- Examines and stores values in the processor's registers
- Examines and stores program code or data in the target system's memory

The Virtual Sourcery Probe is basically the same thing as a real probe and is even built from the same source code tree. The Mentor Embedded Sourcery Probe Debug Server software utilizes the Virtual Sourcery Probe JTAG Transactor API to access the JTAG interface of a Design Under Test (DUT) hardware simulation/emulation model.

Sourcery Probe Supported Processors

The Mentor Embedded Sourcery Probe supports some ARM® processors, MIPS® processors, and PowerPC™ processors. The following tables list the specific processors that are supported.

Table 1-1. Supported ARM Processors

arm7tdmi	arm7t	arm7ts
arm710t	arm720t	arm7ejs
arm920t	arm922t	arm926ejs
arm940t	arm946e	arm966e
arm968e	arm1136j,	arm1136jf
arm1156t2	arm1156t2f	arm1176jz
arm1176jzf	arm11mp	cortex-a8

Table 1-1. Supported ARM Processors (cont.)

a8	cortex-a9	a9
cortex-m3	m3	cortex-m4
m4	cortex-m4f	m4f
cortex-r4	r4	cortex-r5
r5	cortex-r4f	r4f
cortex-r5f	r5f	

Table 1-2. Supported MIPS Processors

bcm3368	3368	bcm4320
4320	bcm4716	4716
bcm4717	4717	bcm4718
4718	bcm4780p	4780p
bcm4785	4785	bcm5350
5350	bcm5352	5352
bcm5354	5354	bcm5365
5365	bcm56218	56218
bcm5834	5834	bcm5836
5836	bcm63268	63268
bcm6338	6338	bcm6345
6345	bcm6348	6348
bcm6352	6352	bcm6358
6358	bcm6368	6368
bcm6816	6816	bcm7115
7115	bcm7125	7125
bcm7312	7312	bcm7315
7315	bcm7318	7318
bcm7405	7405	bcm7420
7420	bcm7422	bcm7425
7425	bcm7550	7550
4kc	4km	4kp

Table 1-2. Supported MIPS Processors (cont.)

4kec	4kem	4kep
5kc	5kf	24kc
24kf	24kec	24kef
74kc	74kf	cnmips

Table 1-3. Supported PPC Processors

mpc8572	mpc8572e	p1022
p2020	p4080	

Sourcery Probe Software Support

Software support for Sourcery Probes is included in most editions of the Mentor® Embedded Sourcery™ CodeBench tool. Install Sourcery CodeBench before attempting to set up or use your Sourcery Probe for the first time.

Sourcery Probe Default Installation Directories

The Sourcery CodeBench installation contains default directories for PowerPC, ARM, and MIPS probes, on both Windows® and Linux®.

Table 1-4 lists the default installation directories.

Table 1-4. Sourcery Probe Default Installation Directories

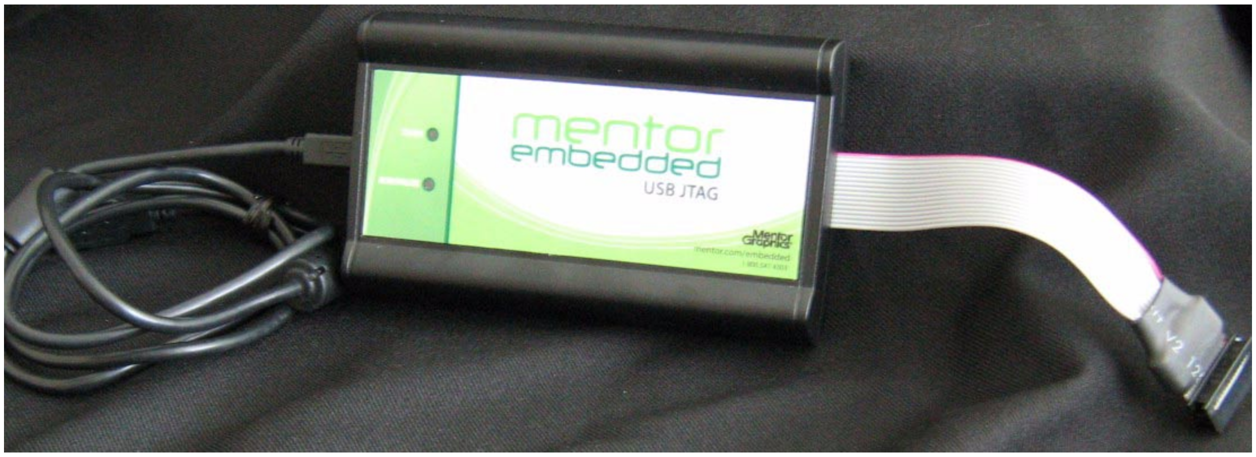
Operating System	Architecture	Installation Directory
Windows	ARM EABI	*<install_dir>/i686-mingw32/arm-none-eabi/mep*
Linux	ARM EABI	*<install_dir>/i686-pc-linux-gnu/arm-none-eabi/mep*
Windows	MIPS	*<install_dir>/i686-mingw32/mips-sde-elf/mep*
Linux	MIPS	*<install_dir>/i686-pc-linux-gnu/mips-sde-elf/mep*
Windows	Power EABI	*<install_dir>/i686-mingw32/power-eabi/mep*
Linux	Power EABI	*<install_dir>/i686-pc-linux-gnu/power-eabi/mep*

Sourcery Probe Personal Models

Sourcery Probe Personal communicates to the host computer via USB, and is also powered by the USB connection. The debug cable conforms to the most common connector type for the given processor architecture, and a series of adapter modules are available for alternate connector types.

Tip: If you have a probe that is labeled Mentor Embedded USB-JTAG Probe, this works exactly the same as Mentor Embedded Sourcery Probe Personal. The only difference is the label on the hardware. Follow the instructions in the *Mentor Embedded Sourcery Probe Personal Hardware Manual*.

Figure 1-1. Sourcery Probe Personal with USB 2.0 Cable



Refer to the *Mentor Embedded Sourcery Probe Personal Hardware Manual* for instructions on:

- Connecting the Sourcery Probe Personal to the host computer
- Installing the USB drivers, and in some cases, configuring communication details
- Connecting the Sourcery Probe Personal to the target debug connector
- Accessing the probe console via the virtual serial port

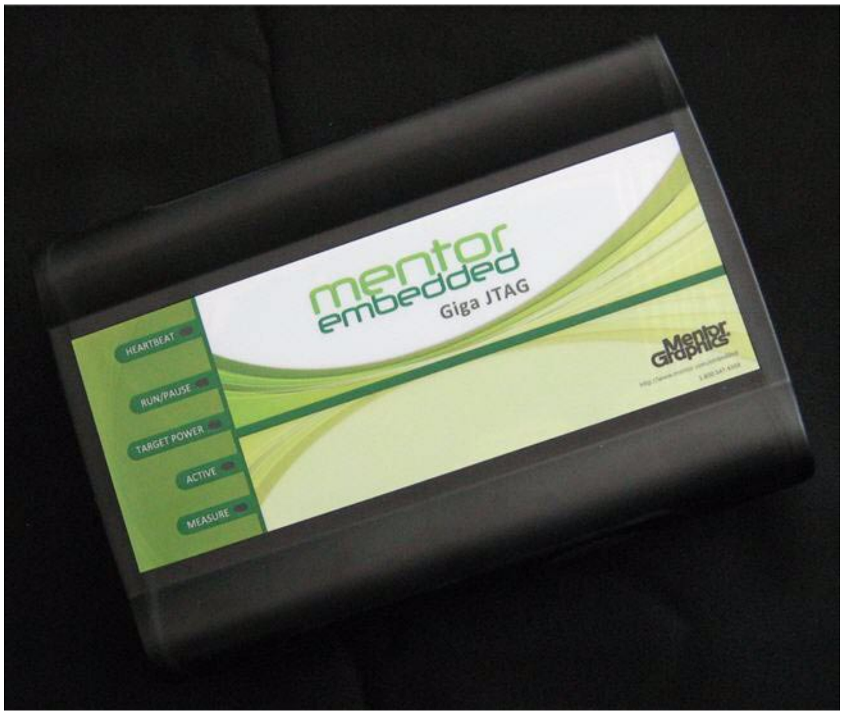
For information on setting up your software, refer to the Sourcery CodeBench *Getting Started Guide*.

Sourcery Probe Professional Models

The Sourcery Probe Professional is powered by an external power supply, and communicates to the host computer via Ethernet. The debug cable is connected to a probe tip, whose debug connector conforms to the most common connector type for the given processor architecture. A series of adapter modules are available for alternate connector types.

Tip: The Mentor Embedded GIGA-JTAG Probe is directly compatible with the Mentor Embedded Sourcery Probe Professional. Follow the instructions in the *Mentor Embedded Sourcery Probe Professional Hardware Manual.*

Figure 1-2. Sourcery Probe Professional



Refer to the *Mentor Embedded Sourcery Probe Professional Hardware Manual* for instructions on:

- Connecting the Sourcery Probe Professional to your network
- Connecting the Sourcery Probe Professional to the target debug connector
- Accessing the probe console via the virtual serial port

For information on setting up your software, refer to the Sourcery CodeBench *Getting Started Guide*.

Chapter 2
Debugging an Application

This section describes how to debug an application with Sourcery CodeBench using a Sourcery Probe.

Note
Except where explicitly stated to the contrary, the term Sourcery Probe refers to all models in the Mentor Embedded Sourcery Probe series.

Debugging an Application with Sourcery CodeBench

Follow the steps in this section to set up a launch configuration for debugging an application using Sourcery CodeBench and a Sourcery Probe.

Prerequisites

- Must have communication with the Sourcery Probe device. For details refer to the following documentation:
  - *Mentor Embedded Sourcery Probe Professional Hardware Manual*
  - *Mentor Embedded Sourcery Probe Personal Hardware Manual*.
  - The Using the Virtual Sourcery Probe chapter in this manual.
- An executable build must exist. Refer to the *Sourcery CodeBench Getting Started Guide* for how to create an executable build.

Procedure

1. In the Sourcery CodeBench IDE, select **Run** > **Debug Configurations...**.
2. Select **Soucery CodeBench Debug**, then click the **New** icon.
3. Under the Main tab, select your project and your application.

Note
If you want to debug an external executable without creating or selecting a project, leave the Project field empty.

4. Under the Debugger tab, select **Sourcery Probe** or **Virtual Sourcery Probe** as the Debug interface.
5. Select the specific probe device you are using.
    - If the probe is discovered, it can be selected from the device list.
    - If some probes are discovered, but not the one you want, select **Manually Configured Probe** and enter its host name or IP address.
    - If no probes are discovered, then enter the probe's host name or IP address.

Note

If the probe you want to use is not shown in the list, click the **Rescan** button (Rescan). If it still is not found, then you must manually enter the IP address or hostname of your probe.

Some common reasons why a probe may not be discoverable include:

- If your probe is not on your current host machine's subnet, or your network adapter is not configured for multi-cast, it might not be discoverable.
- Your probe also might not be discoverable if the routers servicing your network are blocking multi-cast packets, a very common practice among companies.
- Many VPN environments also block multi-cast packets.
- If multi-casting is blocked, you must manually enter the IP address or hostname of your probe.

6. Select the target board.
7. Under Debugger Options select the **Sourcery Probe** tab and enter the appropriate device settings. See Sourcery Probe Debugger Tab Options for a description of these options.
8. Select the **Startup** tab. Use this tab to customize the startup process.
9. By default, CodeBench automatically runs the code up to main() when you launch the debugger. If you instead wish to begin debugging from the program's entry point, then check that option in the Startup tab.
10. Click **Apply**.
11. Click **Debug**.

Related Topics

Creating a Custom Initialization Script Using a Template

Sourcery Probe Target Initialization Scripts

Sourcery Probe Discovery Dialog Box

Sourcery Probe Debugger Tab Options

The Sourcery Probe requires you to set some device options to complete the launch configuration setup.

These options are described in Table 2-1:

Table 2-1. Sourcery Probe Device Options

Option	**Description**
Target init script	Specifies the full path to a target initialization script. To find the list of supplied reference platform initialization scripts, look for files with the *.maj* extension under *i686-mingw32/xxxx/mep/tsp* (Windows host) or *i686-pc-linux/xxxx/mep/tsp* (Linux host), where *xxxx* is the CodeBench package type. Refer to Sourcery Probe Target Initialization Scripts.
CPU ID	Specifies the CPU type on the target board.
Core	Specifies the processor core to which the debugger should attach. NOTE: This option is only applicable to multi-core processors. The integer value must be between 1 and the number of processor cores. See Core_Access_Select.
Endianness	Specifies big or little endian, depending on the byte ordering of the target board. See Trgt_Little_Endian.
Semihosting	Specifies semihosting on or off. The default is on. If semihosting is not in use by the program/application disabling this feature can improve performance.
Log File	This is for the logging operation of MDI and its interface to the probe. It is only required for troubleshooting debug connection problems. NOTE: Be sure to use a fully qualified path for the log file (e.g., *c:\mdi.log*) to make finding it easier. Otherwise, a simple file name or relative path will place the file relative to the current working directory of the Sprite, the location of which is not always obvious.

Related Topics

Debugging an Application with Sourcery CodeBench

Sourcery Probe Target Initialization Scripts

Chapter 3
Sourcery Probe Configuration

This section describes the initialization process when you launch the Sourcery CodeBench debugger and connect to a Sourcery Probe for ARM or MIPS. The initialization process varies slightly, depending on your target system and use case. The different configuration options for the Sourcery Probes are also discussed.

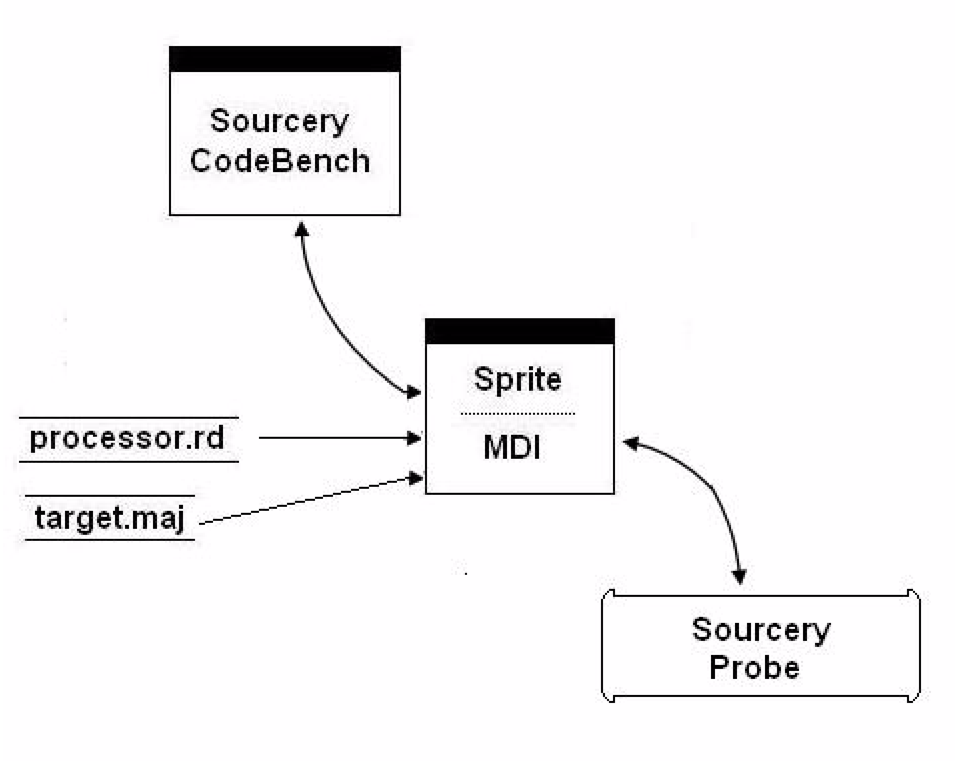
Please refer to Debugging an Application for information on configuring Sourcery CodeBench for use with Sourcery Probe.

Configuration Process

There is a configuration process when using the debugger to connect to a Sourcery Probe.

Figure 3-1 demonstrates the configuration process.

Figure 3-1. Configuration Process



Configuration Files

There are configuration files used in the initialization process.

Table 3-1 describes the configuration files.

Table 3-1. Debugger Configuration Files

processor.rd	The register definition file for the selected processor is automatically read on startup. You may also specify other *.rd* files for custom hardware on your system. (See Register Definition File.)
target.maj	This is a custom target initialization file for managing the details particular to your target board. In some cases, additional script files may be called for more sophisticated initialization scenarios. (See Sourcery Probe Target Initialization Scripts.)

File Search Order

When a file to be opened is specified without specifying a path, or with a relative path name, MDI searches for the file in a list of directories. The same search algorithm is used, with minor variations, in almost all cases.

Relative path names are searched for in the following directories, in order.

Note

Except as noted, after each directory is searched, the debugger checks a subdirectory named *./le* if the target memory system is little-endian, or *./be* if the target memory system is big-endian.

1. The current working directory, except when opening internal files that are expected to be found within the debugger installation directory
2. If a Command or Register Definition file is currently being read, the directory containing that file (but not a *./be* or *./le* subdirectory)
3. The directory containing the MDI shared library
4. Each directory given by the PATH environment variable, in order

For example, if MDI searches for the <*target*>*.maj* command file, it uses the first <*target*>*.maj* file that it finds. If <*target*>*.maj* reads a user-supplied initialization file, the same search order is used to find that file, unless a full path is provided.

Sourcery Probe Target Initialization Scripts

For instructions about working with initialization scripts, use the following list to determine which applies to your use case:

- If you are using the Nucleus® ReadyStart™ tool, the Sourcery Probe initialization script is included in the BSP. Refer to the BSP documentation for specific instructions for your BSP.
- If you are using a supported reference platform, select the appropriate Sourcery Probe Target initialization (*<target>.maj*) file. To find the list of supplied reference platform initialization scripts, look for files with the *.maj* extension under *i686-mingw32/xxxx/mep/tsp* (Windows host) or *i686-pc-linux/xxxx/mep/tsp* (Linux host), where *xxxx* is the CodeBench package type.
- If you are using a custom board based on a supported reference platform, you can adapt the Sourcery Probe Target initialization script for that reference platform to suit your board.

  See Adapting a Reference Board Initialization Script to Your Board.
- If you are using a custom board, you can create a Sourcery Probe Target initialization script by filling in the supplied template script.

  See Creating a Custom Initialization Script Using a Template.

For details on the different components of the initialization script, see Components of Initialization Files. For more information on script files in general, see Command Script Files.

Creating a Custom Initialization Script Using a Template

This section describes how to create a custom initialization script from a raw template to adapt to your custom board.

If your board is based on a supported reference platform, then you should follow the procedure in Adapting a Reference Board Initialization Script to Your Board instead.

Note

The script file *template.maj* provides detailed instructions for editing it to support your specific board. You must follow these instructions to complete your initialization script.

Prerequisite

Locate the Sourcery Probe target initialization files within the Sourcery CodeBench installation.

Procedure

1. Import *template.maj* from the Sourcery CodeBench installation into your project using *File > Import... / General / File System*:

   See Table 1-4 for the installation directory for your specific architecture and OS, then look in the *tsp/_templates* subdirectory.

2. Rename *template.maj* in your project to a more suitable name (*<target>.maj*). It is recommended to use the name of the board because it is a board initialization file.

3. Open *<target>.maj* within Sourcery CodeBench or use a text editor.

4. Carefully read the comment block at the top of the board initialization file. This comment block outlines what needs to be modified to accommodate the details of your board.

5. Edit the file to accommodate your board:

   - The target initialization script must be compatible with the boot code. In general it is best to do a bare minimum of initialization in the script and leave as much as possible to the boot code.

   - Depending on the existing level of support for your target, you should consult different sources for answers to the template questions:

     If there was no prior support for your target anywhere, consult your board documentation.

     If you have access to existing bootloader code, consult the system clocking and memory controller initialization code and derive the script startup code from it. Additionally, consult your board supplier for existing probe initialization scripts.

   - When you get to TODO #4, Memory Controller Setup, consider defining names to help make the script more readable. For example, the AMC_RCR register controls whether the memory is laid out in reboot mode or remap mode in many ARM targets. You can manage this register in several ways:

     - Directly by adding the following command to set the register to 1 through its memory-mapped address:

       ```
       ew 0xFFE00020:P = 0x00000001 //Write AMC_RCR:Issue Re-map command
       ```

     - Set a MON local variable to the register's address (see MON Local Variables and Option References) and then de-reference that variable to set the register:

       ```
       ew $AMC_RCR = 0xFFE00020:P //Set $AMC_RCR to the register address
       ew @$AMC_RCR = 0x00000001  //Write AMC_RCR : Issue Re-map command
       ```

       Note that this method also can only reference memory-mapped registers.

- o Use a Register Definition File to declare the register name and field names, and set the register value by name:

  ```
  ew AMC_RCR = 0x00000001 // Write to AMC_RCR by name
  ```

  Note that if your register names conflict with any program symbols you have loaded, the program symbols have precedence. You can still refer to the register name by adding a "." (dot) in front of the register name.

- o Create and reference a memory-mapped symbol using the `EN Enter Names` command. The disadvantage of this approach is that program symbols take precedence and there is no way to reference this particular symbol in such a case. It also limits references to memory-mapped registers. Example usage:

  ```
  en AMC_RCR = 0xFFE00020:P // Set AMC_RCR to the register address
  ew AMC_RCR = 0x00000001 // Write AMC_RCR : Issue Re-map command
  ```

Related Topics

Components of Initialization Files

Sourcery Probe Target Initialization Scripts

Adapting a Reference Board Initialization Script to Your Board

This section provides the steps required to modify an existing reference board initialization script for your custom board.

If your board is not based on a supported reference platform, then you should follow the procedure described in Creating a Custom Initialization Script Using a Template.

Prerequisites

- A new basic project has been created in Sourcery CodeBench.
- The Sourcery Probe target initialization files have been located within the Sourcery CodeBench installation.

Procedure

1. Import the reference board's *<target>.maj* initialization script from the Sourcery CodeBench installation or Nucleus® BSP package into your product. See Table 1-4 for default installation directory locations, then look in the *tsp* subdirectory.

   Or, if you are using Nucleus ReadyStart, copy the initialization script from the reference board BSP installation directory to your working directory.

2. Rename <*target*>.*maj* file in your working directory to a more suitable name (<*new_target*>.maj). It is recommended to use the name of the board because it is a board initialization file.
3. Open <*new_target*>.*maj* within Sourcery CodeBench or use a text editor.
4. Edit the file to adapt it to the customizations you made to the board relative to the reference platform. Detailed comments in the init script indicate places where changes are likely.

Related Topics

Components of Initialization Files

Sourcery Probe Target Initialization Scripts

Creating a Custom Initialization Script Using a Template

Components of Initialization Files

A typical board initialization file has several separate scripts that can be run at different times for different functions.

Note

A block comment at the top of the file indicates what details need to be filled in, and where to do that. Each section of the initialization script also contains detailed comments on what happens there and what customization, if any, might be required.

Initialization

When the Sourcery CodeBench launches the MDI debug connection, it reads this section of the target initialization script file from the top.

It uses EA (Enter Alias) commands to define new command names that are later used to invoke the command script functions provided later in the script file.

Table 3-2. Initialization File Commands

RTNI	Reset Target with No Initialization
RTI	Reset Target and Initialize
TI	Initialize Target

After these command aliases are defined, the script proceeds to configure the debug interface. Often this requires only a couple of Configuration Options to be set.

Some targets require special probe initialization options. See Special Probe Initialization (ARM and MIPS Only) for details on the probe initialization features normally used during initialization.

When the probe interface has been configured, the script enables the power sensor. (See ice_power_sense in Table 3-9)

Assuming the target is correctly connected and is powered up, the probe detects the target's voltage level and enables the debug interface at that level. At this point the interface and the processor's debug controller are initialized. Normally the processor is stopped and reset when the debug controller is initialized, but if the debugger was launched in a non-intrusive attach mode connection, the processor is stopped, but not reset to allow for inspecting the current state. See R, RP, RT (Reset Processor).

This section of the *template.maj* file also declares the memory configuration table for the board's physical memory map. This controls how the Mentor Embedded Sourcery Probe accesses different memory regions, and helps avoid illegal accesses. See Memory Configuration (ARM and MIPS Only) for more information. The next lines of the file are the place to add additional customization your board might require.

If the debugger was launched in non-intrusive connection mode the initialization script exits; otherwise, it calls one of the other scripts in the file via one of the Initialization File Commands. The correct script to call depends on your particular use case.

DO_RTNI Script

The DO_RTNI script performs a Reset Target type reset.

Note

See R, RP, RT (Reset Processor) with No Initialization (**NI**).

Use the DO_RTNI script to reset the board using the system reset pin on the debug connector (separate from the JTAG reset) and leave the hardware uninitialized. Reset without initialization is recommended when using a boot loader to initialize the board and download the application, or when debugging boot code in ROM or flash memory.

The result depends on the design of your target board and CPU. Certain processors execute some amount of boot code when they exit the reset state despite the presence of a probe. Other boards do not provide a system reset pin on the debug connector and, therefore, cannot be reset using the DO_RTNI script.

DO_RTI Script

The DO_RTI script performs a Reset Target and Initialize.

Note
See R, RP, RT (Reset Processor)

The script first performs a Reset Target type reset to reset the board using the system reset pin on the debug connector (separate from the JTAG reset). After the reset is complete, the script calls the Initialize Target (**TI**) to run your initialization script. Using the DO_RTI script for initialization is recommended when you plan to download the application into RAM using the debugger. However, some targets do not respond well to a hard reset, in which case **TI** is recommended instead of **RTI**.

The result depends on the design of your target board and CPU. Certain processors execute some amount of boot code when they exit the reset state despite the presence of a probe. Other boards do not provide a system reset pin on the debug connector, and, therefore, cannot be reset using the DO_RTI script.

DO_TI

The DO_TI script is used for initializing your memory controller and any additional hardware, as required. If your board boots up normally, you might not need to make any adjustments. However, if you have no boot code (or bad boot code) and want to download code into RAM, then you might need to use the DO_TI script to initialize your memory controller and to set up the control registers.

Note
The target initialization script must be compatible with the boot code. It is best to do a bare minimum of initialization in the script and leave as much as possible to the boot code.

Configuration Options

Many operating parameters of the debug environment are set through configuration options. Some of the configuration options control the behavior of the Mentor Embedded Sourcery Probe, some describe aspects of the target system.

Note
Refer to Configuration Option Table for a comprehensive list of all the configuration options.

Setting Configuration Options

Configuration options can be referred to by their full name, or an abbreviation consisting of the first character from each part of its name. For example, Reset_Address and Ice_Jtag_Clock_Freq can be referred to as **RA** and **IJCF**, respectively.

Some examples are listed here:

```
eo Trgt_Resets_JTAG  = yes       /*Informs the probe if the
                                 target board will forward a system
                                 reset (nSRST) to the JTAG interface
                                 (nTRST),or reset the system only
                                 without resetting the JTAG interface
                                 */

eo Ice_JTAG_Use_RTCLK  = on      /*Selects adaptive clocking mode*/

eo Ice_JTAG_Clock_Freq = 10      /*Specifies the the JTAG clock (TCK)
                                 frequency (if IJUR=OFF)*/

eo ice_power_sense = VREF        /*Enables the target power monitor*/



eo Ice_JTAG_TAP_Select = <num>   /*Controls which TAP on a multi-TAP
                                 daisy chain is associated with this debug
                                 connection */
```

To display a table of all the configuration options and their current settings, enter the DO command with no parameters.

To display a particular option, enter a DO command and specify the option name (or abbreviation).

For a verbose description of an option, use the DOV command.

To display only the option name and value with no commentary, use the DOQ version of the command.

An asterix (*) can be used as a wild card to display all configuration options that exactly match up to the *. For example, the following command displays all configuration options beginning with ice.

```
MON> DO                 // List ALL configuration option settings
MON> DO ice*            // List ALL options startice with "ice"
MON> DOV Ice_Jtag_Tap_Select     // Show details on this one
MON> DOV IJTS           // Show details on Ice_Jtag_Tap_Select
MON> DOQ t*   // Show the names and values of all options starting with T
```

Memory Configuration (ARM and MIPS Only)

The memory configuration (**MC**) table provides the Mentor Embedded Sourcery Probe with details about your memory system. It defines a memory map describing the characteristics of each range in the *physical* address space of the target system.

MC Display (ARM and MIPS only)

Display the **MC** table with the MC command. Issue the **MC** command with no parameters to display the entire memory configuration table. Use an MC command with an address range, but no attribute specifiers to display that part of the table.

MC Attributes Table (ARM and MIPS Only)

The MC attributes provide control of memory access and data width.

Table 3-3 details the **MC** attributes.

Table 3-3. MC Attributes

Attribute	**Settings**	**Description**
Access Method	JAM, DMA, INV	The Sourcery Probe accesses memory either by jamming instructions or using debugger DMA, although not all processors support both modes. Memory regions can also be flagged as invalid in the MC table; the Sourcery Probe will never attempt to access an address flagged as invalid, although it cannot prevent your code from attempting to do so.
Partial Word Access	PWD, PWE	The Sourcery Probe can be set to Partial Word access Enable (**PWE**) or Partial Word access Disable (**PWD**) at particular address regions. This specifies whether the Mentor Embedded Sourcery Probe can perform accesses that are narrower than the actual bus width, as specified with the **DW=***n* setting for that range. When the Mentor Embedded Sourcery Probe attempts to read a partial word from an address where partial word access is disabled, it will first read a data-width-sized word, then extract the desired part. For writes, it will perform a read-modify-write operation. This is optimized so that one command to read several bytes in the same word only accesses the target once, and writing several bytes to the same word performs one read and one write.

Table 3-3. MC Attributes (cont.)

Attribute	Settings	Description
Read-Only	RO, RW	The Read-Only flag controls whether the Sourcery Probe is allowed to write to the memory range. When set to Read-Only (**RO)** mode, the Mentor Embedded Sourcery Probe can read from memory within the range, but will never write within the range. In Read-Write (**RW)** mode, the Mentor Embedded Sourcery Probe can read or write within the range.
Data Width	DW=8, DW=16, DW=32, DW=64	The Data Width option defines the maximum size data transfer the Mentor Embedded Sourcery Probe can perform in the given range. Access requests that are wider than this setting are performed by reading or writing a block of data-width-sized objects. This option also controls the transfer size used by the Mentor Embedded Sourcery Probe in **PWD** mode transfers. The Data Width attribute should normally be set to the natural bus width of your processor. It should only be reduced if your memory controller has trouble with wide accesses.

Setting MC Attributes (ARM and MIPS Only)

The **MC** command can be used to set an individual attribute, or multiple attributes, for the specified memory region. When an **MC** command is entered, only those attributes specified in the command are changed. All other attributes remain unaffected.

Before you set up the memory configuration table, you must be familiar with your system's physical memory map. Specifically:

- Where is your ROM, and how big is it?
- Where is your RAM, and how big is it?
- What peripherals do you have, where are they mapped, and are they byte accessible?
- What other memory-mapped resources are there, and how are they accessed?

When you have a concise representation of your memory system, you can enter it into the Mentor Embedded Sourcery Probe memory configuration table. Add the appropriate **MC** commands to your target initialization file, where noted in the template. The following are examples of additions to the file:

```
MC *:P,INV                      /* Flag all physical memory as invalid */
MC 0:P FFFFFF:P,DMA             /* Set first 16MB to DMA access */
MC 10000000:P 1000FFFF:P,JAM    /* Select JAM mode for this range */
MC *:P, PWE                     /* Enable partial word access for all
                                   physical memory */
MC 10000000:P 1000FFFF:P,PWD    /* Disable partial word access for
                                   selected range */
MC 0:P FFFFF:P, RO, DW=8        /* First Meg is 8-bit read-only */
```

Note

Because the MC table describes your *physical* memory environment, *physical* addresses must be used when setting MC attributes. Physical addresses are specified by appending *:P* to the address value.

Note

As MC commands are read or entered, the Mentor Embedded Sourcery Probe collates the input ranges into one map, spanning the entire address space, with no holes or overlaps.

Sample MC Table

For this example, assume the memory model consists of the following regions:

- 512k of read-only flash memory at 0x1FC00000 in the physical memory space.
- 1MB of RAM starting at address 0 in the physical memory space.
- 256k of internal scratchpad RAM at 0x10000000 in the physical memory space.

- Peripherals are located at 0x18000000 in the physical memory space, and do not support byte writes.

Assume that your processor supports DMA to external memory, but not to internal scratchpad RAM. You want to prevent inadvertent accesses (by the Mentor Embedded Sourcery Probe) to invalid memory regions. The following memory configuration commands describe the system:

```
MC *:P, INV  /* Invalidate all memory first, then configure valid ranges*/
MC 00000000:P 000FFFFF:P, DMA, PWE      /* DRAM */
MC 10000000:P 1003FFFF:P, JAM, PWE      /* Scratchpad RAM*/
MC 18000000:P 18FFFFFF:P, DMA, PWD      /* Peripherals */
MC 1FC00000:P 1FC7FFFF:P, DMA, PWD, RO  /* Flash */
```

The first line defines all memory as invalid, thereby erasing any previous mappings. This prevents the Mentor Embedded Sourcery Probe from attempting to access any address (although it cannot prevent your program from doing so, while it is running).

The second line sets a 1MB region, starting at 0, that can be accessed by DMA, and supports partial word accesses (PWE). This area is no longer restricted. If your CPU does not support DMA, then select JAM instead.

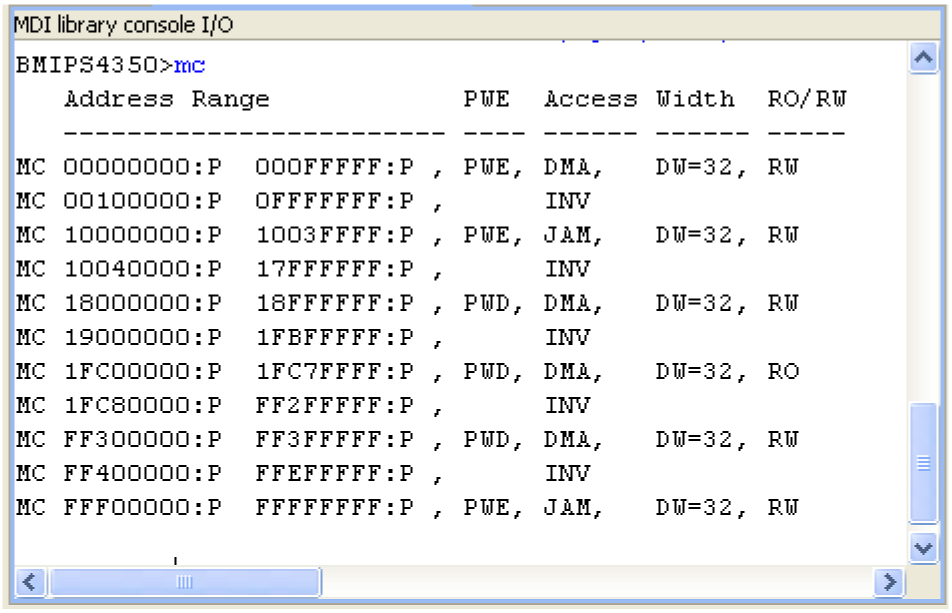
The third line represents the internal scratchpad RAM, which must be accessed by jamming load and store instructions (JAM), because DMA is not supported in this area for this example. Partial word accesses are enabled (PWE) in this area.

The peripheral area is shown on line 4; DMA access mode is enabled. However, partial word accesses are disabled (PWD), since these hypothetical peripherals do not support byte writes.

The boot ROM is shown on line 5; DMA is enabled, and partial word accesses are disabled (PWD). This range is also marked as read-only (RO), because flash is not directly writable.

The memory configuration commands would result in the memory configuration table as shown in Figure 3-2.

Figure 3-2. Memory Configuration Table



MDI library console I/O

```
BMIPS4350>mc
   Address Range              PWE  Access Width  RO/RW
   -------------------------  ---- ------ ------ -----
MC 00000000:P   000FFFFF:P , PWE, DMA,   DW=32, RW
MC 00100000:P   0FFFFFFF:P ,      INV
MC 10000000:P   1003FFFF:P , PWE, JAM,   DW=32, RW
MC 10040000:P   17FFFFFF:P ,      INV
MC 18000000:P   18FFFFFF:P , PWD, DMA,   DW=32, RW
MC 19000000:P   1FBFFFFF:P ,      INV
MC 1FC00000:P   1FC7FFFF:P , PWD, DMA,   DW=32, RO
MC 1FC80000:P   FF2FFFFF:P ,      INV
MC FF300000:P   FF3FFFFF:P , PWD, DMA,   DW=32, RW
MC FF400000:P   FFEFFFFF:P ,      INV
MC FFF00000:P   FFFFFFFF:P , PWE, JAM,   DW=32, RW
```

Note
The Mentor Embedded Sourcery Probe may coerce the settings of certain memory ranges to meet access method restrictions imposed by the target processor. For example, if internal memory-mapped registers are not accessible through DMA, the Mentor Embedded Sourcery Probe will keep such areas set to JAM mode.

Register Definition File

When Sourcery CodeBench opens an MDI connection, the MDI library automatically reads the register definition file corresponding to the CPU type you selected in the launch configuration properties. The register definition file is where the CPU and system coprocessor registers are defined.

You can also use a custom register definition file to add your own definitions for application-specific co-processor registers and memory-mapped registers. This allows you to access special registers by name with MON commands (see MON Command Language) in the MDI window or Mentor Embedded Sourcery Probe script files instead of requiring you to remember their addresses. In addition to assigning names to the registers, bit fields within the registers can be defined so that the bit fields can be viewed or set by name.

Note
Registers defined in this way are known only to the MON command line and scripts (see MON Command Language).

Register Definition File Creation

After preparing your target initialization script follow the procedure below to create and use register definition files for your custom hardware.

See the Sourcery Probe Target Initialization Scripts section for details.

Procedure

1. Import the template file, named *template.rd*, into your project using **File > Import... / General / File System**, in the same folder that contains your Sourcery Probe target initialization file.

Note
It is recommended to rename the file to associate it with the hardware it describes.

Tip: See Table 1-4 for default installation directory locations, then look in the *tsp/_templates* subdirectory.

2. Add your register details with a text editor or with CodeBench, as described in Register Definition File Format.

3. Add the following command to your custom initialization file to read your register definitions:

```
fr rd filename
```

Tip: You can use the same technique to define "debugger local" register names and field breakdowns. In this case, the debugger allocates space for the register in its own local memory, rather than at a specified target register or memory address. One use for debugger local registers is to enable fields to be used with write-only hardware resources. This is for the case where the read-modify-write sequence occurs when a value is assigned to a register field that would not work with the real hardware. A debugger local register can be defined with the field breakdown. Because the fields can be assigned values individually, the whole register can be copied to the real target register in one operation.

Register Definition File Format

Define new register names using the following syntax:

```
REG = reg_name [offset space_name] byte_size [SEQ first last
     obj_inc [inc]]
```

Table 3-4 describes the variable definitions.

Table 3-4. Variable Descriptions

Variable Name	Description
reg_name	Identifier giving the name of the register being defined. If the name begins with '$', it defines a "debugger local" register and the offset and the space_name are omitted.
offset	Decimal number giving the register's byte offset within the specified space. For real register spaces, the offset is the register number times the register size. For memory spaces, the offset is the byte address.
space_name	One of the keywords from the following list giving the register file or memory space for the register. See Table 3-6 and Table 3-7.
byte_size	Size of the register in bytes (1, 2, 4. or 8).
first	Decimal number giving the first value to append to reg_name to form a sequence of names. Sequences make it easy to represent a consecutive set of like-named registers (for example, r0..r31).
last	The last number in the sequence (see first).
obj_inc	Decimal number giving the "stride". It is normally *1*, but can be set to a higher value when the registers in the sequence are actually addressed as every Nth register in the space. For each register in the sequence, the amount added to the offset to form the address is the register sequence number * *byte_size*.
inc	Decimal number giving the amount by which to increment the register number for each name in the sequence. If not specified, the default is 1.

For example: You want a sequence of four 8 bit registers mapped to physical memory at **0**, with each register in the low byte of successive machine words (32 bits). To name these registers *z1*, *z3*, *z5*, etc., the definition syntax is:

```
REG=z 0x0 MEMORY_P 1 SEQ 1 7 4 2
```

Register Field Definition File Format

Registers can also be broken down into displayable fields. Any previously defined register or register sequence can be set up as *field encoded*. If a field breakdown is provided for a register sequence, the fields apply to every register in the sequence. Fields of more than one bit are displayed as *field_name=hexadecimal_value*. One bit fields are displayed as an uppercase or lowercase *field_name* where uppercase means a **TRUE** or **1** value.

```
REG_FIELD = reg_name field_spec [, field_spec]
```

Table 3-5 shows the REG_FIELD variable descriptions.

Additional Notes:

- Long REG_FIELD definitions can be continued on more than one line. If a line ends with a comma, list item processing continues on the next line.
- Include files are supported to allow common processor elements to be placed in one file. The INCLUDE command begins reading from the referenced file and returns to the calling file when done. Nested include files are allowed.

```
INCLUDE "filename"
```

Table 3-5. REG_FIELD Variables

Variable Name	Definition
reg_name	The name previously defined through a REG statement. Note that for sequence registers a full sequence register name must be given (including the number).
field_spec	field_name high_bit low_bit [, field_spec]
field_name	An ident providing the name of the field.
high_bit	A decimal number in the bit range of the given register. Must be >= low_bit.
low_bit	A decimal number in the bit range of the given register. Must be <= high_bit.

Sample Register Definition file

The following example demonstrates a definition for some memory-mapped registers (common in hardware designs).

```
// Define a register named 'dev_a_ctrl' and associate its address, space,
and size
REG       = dev_a_ctrl  0xFF00A000 PHYSICAL 4

// Define the names and bit positions of fields within the 'dev_a_ctrl'
register
REG_FIELD = dev_a_ctrl  status 2 0, lock 3 3

// Define additional registers
REG = dev_a_data1 0xFF00A004 MEMORY 4
REG = dev_a_data2 0xFF00A008 MEMORY 4

// <eof>
```

Predefined Spaces for ARM

There exists pre-defined address spaces for the ARM processor.

Table 3-6 details pre-defined spaces for ARM.

Table 3-6. Predefined Spaces for ARM

X – Supported, RSVD – Reserved, NS – Not Supported, NA – Not Applicable.

Space Name	Description	ARM7/9/11	Cortex-A	Cortex-M
MEMORY	Virtual Memory	X - ARM9/11 NS - ARM7	X	X[1]
PHYSICAL	Physical Memory	X	X	X
CRNT	General Registers r0 - r15	X	X	X Note A
USER	User/System mode registers	X	X	NA
SVC	Supervisor mode registers	X	X	NA
IRQ	Interrupt mode registers	X	X	NA
FIQ	Fast Interrupt mode registers	X	X	NA
ABORT	Abort mode registers	X	X	NA
UNDEF	Undefined exception mode registers	X	X	NA
STATUS (non-Cortex-M)	cpsr, spsr {svc, abort, undef, irq, fiq}	X	X	NA
STATUS (Cortex-M only)	xpsr, apsr, ipsr, epsr, PRIMASK, FAULTMASK, BASEPRI, CONTROL Note B	NA	NA	X
COPROC0	CoProcessor 0 registers	ARM7-Implementation Defined[2] ARM9-Implementation Defined[3] ARM11-Vendor specific[4]	Vendor specific	Implementation defined

Table 3-6. Predefined Spaces for ARM (cont.)

X – Supported, RSVD – Reserved, NS – Not Supported, NA – Not Applicable.

Space Name	Description	ARM7/9/11	Cortex-A	Cortex-M
COPROC1	CoProcessor 1 registers	ARM7-Implementation Defined ARM9-Implementation Defined ARM11-Vendor specific	Vendor specific	Implementation defined
COPROC2	CoProcessor 2 registers	ARM7-Implementation Defined ARM9-Implementation Defined ARM11-Vendor specific	Vendor specific	Implementation defined
COPROC3	CoProcessor 3 registers	ARM7-Implementation Defined ARM9-Implementation Defined ARM11-Vendor specific	Vendor specific	Implementation defined
COPROC4	CoProcessor 4 registers	ARM7-Implementation Defined ARM9-Implementation Defined ARM11-Vendor specific	Vendor specific	Implementation defined
COPROC5	CoProcessor 5 registers	ARM7-Implementation Defined ARM9-Implementation Defined ARM11-Vendor specific	Vendor specific	Implementation defined

Table 3-6. Predefined Spaces for ARM (cont.)

X – Supported, RSVD – Reserved, NS – Not Supported, NA – Not Applicable.

Space Name	Description	ARM7/9/11	Cortex-A	Cortex-M
COPROC6	CoProcessor 6 registers	ARM7-Implementation Defined ARM9-Implementation Defined ARM11-Vendor specific	Vendor specific	Implementation defined
COPROC7	CoProcessor 7 registers	ARM7-Implementation Defined ARM9-Implementation Defined ARM11-Vendor specific	Vendor specific	Implementation defined
COPROC8	CoProcessor 8 registers	ARM7-Implementation Defined ARM9-Implementation Defined ARM11-Vendor specific	RSVD	RSVD
COPROC9	CoProcessor 9 registers	ARM7-Implementation Defined ARM9-Implementation Defined ARM11-Vendor specific	RSVD	RSVD
COPROC10	CoProcessor 10 registers	ARM7-Implementation Defined ARM9-Implementation Defined ARM11-NS	X	RSVD

Table 3-6. Predefined Spaces for ARM (cont.)

X – Supported, RSVD – Reserved, NS – Not Supported, NA – Not Applicable.

Space Name	Description	ARM7/9/11	Cortex-A	Cortex-M
COPROC11	CoProcessor 11 registers	ARM7-Implementation Defined ARM9-Implementation Defined ARM11-NS	X	RSVD
COPROC12	CoProcessor 12 registers	ARM7-Implementation Defined ARM9-Implementation Defined ARM11-Vendor specific	RSVD	RSVD
COPROC13	CoProcessor 13 registers	ARM7-Implementation Defined ARM9-Implementation Defined ARM11-Vendor specific	RSVD	RSVD
COPROC14	CoProcessor 14 registers	ARM7-Implementation Defined ARM9-Implementation Defined ARM11-Not user accessible[5]	X	RSVD
COPROC15	CoProcessor 15 registers	ARM7-Implementation Defined ARM9-X ARM11-X	X	RSVD

1. A virtual address (VA) is always equal to a physical address (PA).
2. For COPROC0-COPROC15 - RD files need to be updated with descriptions of implementation specific registers
3. For COPROC0-COPROC14 - RD files need to be updated with descriptions of implementation specific registers
4. For COPROC0-COPROC13 - RD files need to be updated with descriptions of vendor specific registers

5. Reserved for the probe to access the debug controller

Note A

R13 Stack pointer register – there are two stacks supported in ARMv7-M, each with its own (banked) stack pointer register:

- Main stack – SP_main
- Process stack – SP_process.

Note B

xPSR = APSR | IPSR | EPSR

Predefined Spaces for MIPS

There exists pre-defined address spaces for the MIPS processor.

Table 3-7 details pre-defined spaces for MIPS.

Table 3-7. Predefined Spaces for MIPS

Space Name	Description
MEMORY	Virtual Memory
PHYSICAL	Physical Memory
GR	General Registers r0 - r31
MR	mdhi, mdlo
CP0_CTL	Some newer MIPS32 chips use this space
CP0_GEN	Coprocessor control register (cause, sr, etc)
CP1_CTL	Floating point control
CP1_GEN	Floating point
CP2_CTL	CP2 Typically not used
CP2_GEN	CP2 Typically not used
CP3_CTL	Mips I/II architecture chips only
CP3_GEN	Mips I/II architecture chips only
ICT	Instruction Cache tags
DCT	Data Cache tags
TLB	TLB registers 0..?

Predefined Spaces for PowerPC

There exists pre-defined address spaces for the PowerPC processor.

Table 3-8 details pre-defined spaces for Power PC.

Table 3-8. Predefined Spaces for PowerPC

Space Name	Description
MEMORY	Virtual Memory
PHYSICAL	Physical Memory
INSTR_MEM	Virtual Memory Accessed as Instruction
GR	General Registers r0 - r31
SPR	Special Purpose Registers spr0 - spr1023
CR	Condition Register cr
MSR	MSR Register msr
FPSCR	Floating Point Control Registers fpscr
TLB0	Registers l2mmu_tlb0 - l2mmu_tlb511
TLB1	Registers l2mmu_cam0 - l2mmu_cam15 (end various with core type)

Configuration Option Table

This section describes the Mentor Embedded Sourcery Probe configuration options.

Note
Not all options are supported for all Mentor Embedded Sourcery Probe models, target types, or debugger environments. The DO command lists all available options that are currently available. Table 3-9 lists the options for all environments.

Configuration options can be referred to by their full name, or an abbreviation consisting of the first character from each part of its name. For example, Reset_Address and Ice_Jtag_Clock_Freq can be referred to as RA and IJCF, respectively. Where an abbreviation exists for an option in Table 3-9, the abbreviation is noted in parenthesis: Ice_Jtag_Clock_Freq (IJCF).

Table 3-9. Configuration Options

Option	CPU Architecture	Valid values	Default
Core_Access_Select		**0-32**	**0**
The **Core_Access_Select** option selects the CPU core to debug with this connection. Cores are numbered 1-N, and a setting of 0 means not connected to any core. When a core is selected, the Ice_Jtag_Tap_Select (if available) option is automatically set to the TAP position associated with the selected core. **NOTE:** This option is not supported by all CPU types. In many cases Ice_Jtag_Tap_Select is used to select the CPU core to debug.			
Calling_Convention	**MIPS Only**	**n32, o32, o64**	**n32**
The **Calling_Convention** option allows you to tell MON which calling convention was followed by the compiler for the program under test. **o32** is the original MIPS standard R3000 calling convention. **n32** is the newer MIPS standard for R4000 (MIPS 3 ISA and later) processors with 32 bit pointers. **o64** refers to the calling convention used by many GNU compilers for the R4000.			
ccs_timeout = 15		**1 - 120**	**15**
Specifies the CCS timeout period in seconds. If the target does not respond in the provided time-interval, you receive a CCS timeout error.			
Ice_Debug_Boot		**on, off**	**on**
If the **Ice_Debug_Boot** (**idb**) option is **on**, the Sourcery Probe will configure the processor to enter debug mode immediately when reset. If the **Ice_Debug_Boot** (**idb**) option is **off**, the processor will execute code from the reset vector until the Sourcery Probe halts it and takes control. **NOTE:** This option is not available on all processors, because only certain processors provide both modes. On many processors, the behavior upon reset depends on how the target reset and JTAG reset circuits are implemented on your board, and how the Trgt_Resets_JTAG option is set.			
Ice_Jtag_Clock_Freq		**0.003 - 44 (Sourcery Probe Personal)** **0.002 - 100 (Sourcery Probe Professional)**	**10**
This option sets the JTAG clock frequency driven on the JTAG clock (TCK) pin, in MHz. A setting of 12.5 means 12.5MHz, and a setting of 0.250 means one quarter of a MHz, or 250kHz. **NOTES:** The Ice_Jtag_Clock_Freq setting has no effect when Ice_Jtag_Use_Rtclk is on. The frequency setting is adjusted to the nearest valid frequency supported by the probe.			

Table 3-9. Configuration Options (cont.)

Option	CPU Architecture	Valid values	Default
Ice_Jtag_Tap_Count		**0 - 1024**	**0**
This option lists the number of devices (TAP controllers) detected on the JTAG scan chain. If there is more than one device on the chain, then the Ice_Jtag_Tap_Select (**ijts**) option must be set to select the CPU to which the Sourcery Probe should connect. **NOTE:** This option is only valid after power has been detected. (see Ice_Power_Sense**)**			
Ice_Jtag_Scan_Freeze		**on, off**	**off**
The Ice_Jtag_Scan_Freeze option controls how the JTAG state machine suspends a scan operation which does not return to RunTest/Idle. When Ice_Jtag_Scan_Freeze is off (the default), it suspends by moving to an IR/Pause or DR/Pause state. When the Ice_Jtag_Scan_Freeze option is on, the JTAG state machine suspends by remaining in an IR/Scan or DR/Scan state while freezing TCK. **Off** is the recommended setting for this option, but **on** is required in certain special cases.			
Ice_Jtag_Tap_Select		**0 - 1024**	**1**
If there are multiple devices (TAP controllers) on the JTAG interface, the Ice_Jtag_Tap_Select option selects the CPU the Mentor Embedded Sourcery Probe will control in this debug session. Devices are numbered 1 to N, where N is the number of JTAG controllers (see Ice_Jtag_Tap_Count). Device 1 is connected to the Sourcery Probe's TDO signal, and device N is connected to its TDI signal. On the Virtual Sourcery Probe, setting this option also initiates the JTAG initialization sequence which normally happens when Ice_Power_Sense is turned on with physical probes.			
Ice_Jtag_Use_Rtclk		**on, off**	**off**
The **Ice_Jtag_Use_Rtclk** option enables or disables adaptive clocking mode on the JTAG interface. This option should be set to **on** prior to setting the Ice_Power_Sense option if the RTCLK signal on the JTAG connector is required by your target system. Otherwise, it should be **off**. With the Virtual Sourcery Probe, the RTCK mode must be set before the Ice_Jtag_Tap_Select option is set.			
Ice_Mem_Block		**0x0 - 0xfffffffc**	**0x00000000**
The **Ice_Mem_Block** (**imb**) option can be used to specify a 1k byte block of RAM on the target that is reserved for Sourcery Probe. When set to a non-zero value, Sourcery Probe can use this memory block to optimize certain debug services. The Sourcery Probe must be able to read, write, and execute code from the specified virtual address at all times, and it must be located in an uncached region of the address space (kseg1 on MIPS processors). When set to 0, the Sourcery Probe does not rely on any target memory for implementing debug services.			
Ice_Multi_Session		**off, on**	**off**
Allows the Sourcery Probe to accept multiple simultaneous debugger connections. When debugging multiple processors on the same JTAG chain, the first debugger session must set this option on before additional connections are attempted.			

Table 3-9. Configuration Options (cont.)

Option	**CPU Architecture**	**Valid values**	**Default**
Ice_Power_Sense		**off, vref**	**off**
The Ice_Power_Sense option controls the target power monitor. Setting this option **off** disables the target debug interface. Setting it to **vref** enables the target power monitor. When target power is detected, the debug interface is initialized. If one TAP is detected, or if the TAP selection was preset in advance, then the debug connection to the CPU is established.			
Ice_Reserved_Hwbps		**0 - 2**	**0**
The Ice_Reserved_Hwbps option sets the number of hardware breakpoint resources reserved by the Sourcery Probe for internal use. On some processors the Sourcery Probe needs to use hardware breakpoint resources to perform other debug functions (such as stepping in ROM). In such cases, it normally reserves breakpoint resources for its most commonly needed internal purposes, and reduces the number of hardware breakpoints that you can enable at one time accordingly. Setting this option to 0 allows you to enable the maximum number of hardware breakpoints supported by the processor. If the Sourcery Probe is not able to allocate a breakpoint resource when needed, then it will not run or step until you free up a breakpoint resource by deleting or disabling one of your hardware breakpoints. For processors that implement separate resources for code and data hardware breakpoints, this issue (and option) applies only to code breakpoints.			
Ice_Reset_Cp15_Cntrl	**ARM**	**0x0 - 0xffffffff**	**0x0**
The **Ice_Reset_Cp15_Cntrl** (**ircc**) option specifies the value assigned to the coprocessor 15 control register upon reset. The value specified takes hardware strapping options of your target board into account.			
Ice_Reset_Delay		**0 - 30000**	**1000**
The Ice_Reset_Delay option controls the minimum time delay after system reset before the Sourcery Probe continues. The time is specified in milliseconds.			
Ice_Reset_Output		**on, off**	**off**
The **Ice_Reset_Output** (**iro**) option controls whether the Sourcery Probe asserts its reset output signal when a Reset (**R**) command or Load (**L**) command is issued by the debugger. This option does not affect the operation of the Reset Processor (**RP**) or Reset Target (**RT**) command.			
Ice_Reset_Time		0 - 30000	250
The Ice_Reset_Time option controls the minimum time that the Sourcery Probe keeps the system reset pin asserted during a Reset Target operation. The time is specified in milliseconds.			
Ice_Step_Masks_Ints	**MIPS**	**off, on**	**off**
If the **Ice_Step_Masks_Ints** (**ismi**) option is **on**, then interrupts are masked while single stepping. Otherwise interrupts are not masked while stepping, and the processor steps into the interrupt handler if an interrupt is pending.			

Table 3-9. Configuration Options (cont.)

Option	CPU Architecture	Valid values	Default
Ice_Shutdown_On_Exit		**yes, no**	**no**
(Restricted to MESP/Virtual) When quitting from MON or CodeBench, VMAJIC and the Veloce emulation job remain running, allowing the debugger to be restarted for a new debug session. However, setting this option to **yes** allows you to shut down VMAJIC and the Veloce emulation job from the debugger side when quitting from MON or CodeBench.			
Load_Absolute_Syms		**on, off**	**off**
The Load_Absolute_Syms option specifies that absolute-valued symbols are included when MON loads a program.			
Load_Entry_Pc		**on, off**	**on**
If the Load_Entry_Pc option is set to **on** when a program is loaded with the MON Load (**L**) command, then the PC register is set to the program's entry point address. If multiple programs are loaded via the same MON Load (**L**) command, the entry point is taken from the first referenced program.			
Reset_Address	**ARM**	**0x0 - 0xffffffff**	**0x00000000**
The **Reset_Address** option controls the initial program pointer (PC) after a reset operation is issued through the debugger. This allows the reset vector to be redirected into RAM. When set to the actual reset vector, no special processing takes place. When set to any other value, the debugger sets the (conceptual) PC as specified by this option after each reset or download command. Note that MON normally performs a reset whenever the program is downloaded with the **L** command, unless inhibited with the Reset_At_Load option.			
Reset_Address	**MIPS**	**0xffffffffa0000000 - 0xffffffffbfffffff**	**0xffffffffbfc00000**
The **Reset_Address** option controls the initial program pointer (PC) after a reset operation is issued through the debugger. This allows the reset vector to be redirected into RAM. When set to the actual reset vector, no special processing takes place. When set to any other value, the debugger sets the (conceptual) PC as specified by this option after each reset or download command. Note that MON normally performs a reset whenever the program is downloaded with the **L** command, unless inhibited with the Reset_At_Load option. **NOTE:** The **reset_address** must be within kseg1.			
Reset_At_Load		**on, off**	**on**
If the **Reset_At_Load** option is set to **on**, the processor is reset before the program is downloaded with the MON Load (L) command. Otherwise, the program is immediately downloaded to the target. See Ice_Reset_Delay, Reset_Address,(ARM Only) and Reset_Address (MIPS Only) for more information on the reset operation. See also Load_Entry_Pc.			
Semi_Hosting_Enabled		**on, off**	**on**
Set the Semi_Hosting_Enabled option to **on** to enable system calls (open, close, read, write) from your program to be serviced via the debugger. For ARM, see also Top_Of_Memory. Note: This option should be **off** when debugging programs using the CodeBench-hosted mode feature.			

Table 3-9. Configuration Options (cont.)

Option	CPU Architecture	Valid values	Default
Semi_Hosting_Vector	**ARM**	**0x0 - 0xfffffffc**	**0x00000008**
The Semi_Hosting_Vector option sets the vector at which semi-hosting calls are intercepted.			
Sym_Delta		**0x0 - 0xffffffff**	**0xffff**
The Sym_Delta option is the maximum offset for the symbol+offset display.			
Top_Of_Memory	**ARM**	**0x0 - 0xffffffff**	**0x0**
Set the Top_Of_Memory option to the 32-bit word-aligned address at the top of your RAM space. This option is used by the ARM Semi-Hosting library for stack initialization. (see Semi_Hosting_Enabled)			
Trgt_Cache_Type		**unknown, none, unified, instruction, data, separate**	**unknown**
The **Trgt_Cache_Type** (**tct**) option reports the type of primary cache(s) provided by the CPU, if it is known.			
Trgt_Cpu_State		**run, halt, sleep, doze, off, disco**	**halt**
The **Trgt_Cpu_State** (**tcs**) option reports the state of the CPU.			
Trgt_Dcache_Linesize		**0 - 1024**	**0**
The **Trgt_Dcache_Linesize** (**tdl**) configuration option reports the size in bytes of each line in the CPU's primary data cache, if it is known.			
Trgt_Dcache_Memsize		**0x0 - 0xffffffff**	**0x0**
The **Trgt_Dcache_Memsize** (**tdm**) option reports the size in bytes of the CPU's primary data cache, if it is known.			
Trgt_Dcache_Sets		**0 - 1024**	**0**
The **Trgt_Dcache_Sets** (**tds**) option reports the number of sets in the CPU's primary data cache, if it is known.			
Trgt_Icache_Linesize		**0 - 1024**	**0**
The **Trgt_Icache_Linesize** (**til**) option reports the size in bytes of each line in the CPU's primary instruction or unified cache, if it is known.			
Trgt_Icache_Memsize		**0x0 - 0xffffffff**	**0x0**
The **Trgt_Icache_Memsize** (**tim**) configuration option reports the size in bytes of the CPU's primary instruction or unified cache, if it is known.			
Trgt_Icache_Sets		**0 - 1024**	**0**
The **Trgt_Icache_Sets** (**tis**) option reports the number of sets in the CPU's primary instruction or unified cache, if it is known.			

Table 3-9. Configuration Options (cont.)

Option	CPU Architecture	Valid values	Default
Trgt_Little_Endian		**on, off**	**off**
Setting **Trgt_Little_Endian** to **on** indicates that your target system has a little-endian memory architecture. The initial value of this option is set with the Sourcery CodeBench / Sourcery Probe Launch configuration, and it is not recommended to change this option manually via MON.			
Trgt_Resets_Jtag		**no, yes**	**no**
The **Trgt_Resets_Jtag** (**trj**) option specifies whether a target system reset also causes a JTAG reset. Although it is not recommended, some target systems reset the JTAG interface when the system reset pin on the debug connector is asserted. The **Trgt_Resets_Jtag** option must be **yes** in this case. For target systems where system reset does not cause a JTAG reset, trgt_resets_jtag should be **no**. See also Ice_Reset_Delay.			
Vector_Catch (ARM7, ARM9, ARM11)	**ARM7, ARM9, ARM11**	**0xFF**	**0x3B**
The Vector_Catch option for ARM processors specifies a bit mask selecting exception vectors to be trapped. See also Vector_Catch (Cortex-A) for behavioral differences with the Cortex-A processor. Some processors provide special hardware for trapping these vectors. For other processors, the selected vectors are trapped by setting breakpoints. Note that software breakpoints can only be set when the vectors are in RAM. This feature should be disabled (set to 0) when debugging code that begins at address 0.			

Bit Value	Default Setting	Exception Name	Exception Vector Address
0x001	on	Reset	0x00
0x002	on	Undefined instruction	0x04
0x004	off	SWI	0x08
0x008	on	Reserved	0x0C
0x010	on	Data Abort	0x10
0x020	on	Prefetch Abort	0x14
0x040	off	IRQ interrupt	0x18
0x080	off	FIQ interrupt	0x1C

Table 3-9. Configuration Options (cont.)

Option	CPU Architecture	Valid values	Default
Vector_Catch (Cortex-A)	**Cortex-A**	**0x0 - 0xffffffff**	**0x0**

The Vector_Catch option for Cortex-A processors specifies that selected exception vectors are to be watchpointed by the probe. See also Vector_Catch (ARM7, ARM9, ARM11) for behavioral differences with the ARM processors and Vector_Catch (Cortex-M) for the Cortex-M processors. When a watchpoint occurs WFAR contains the address of the instruction causing it plus 8 for ARM mode, or plus 4 for THUMB mode. This feature should be disabled (set to 0) when debugging code that begins at 0.
The bit mask is defined as (decimal bit, hex field bit, and cause):

Secure World			**Secure Monitor**			**Nonsecure World**		
Bit	**Field**	**Cause**	**Bit**	**Field**	**Cause**	**Bit**	**Field**	**Cause**
0	01	Reset	10	0400	SMC	25	02000000	Undef Instr
1	02	Undef Instr	11	0800	Prefetch Abt	26	04000000	SVC
2	04	SVC	12	1000	Data Abt	27	08000000	Prefetch Abt
3	08	Prefetch Abt	13	4000	IRQ	28	10000000	Data Abt
4	10	Data Abt	14	8000	FIQ	30	40000000	IRQ
6	40	IRQ				31	80000000	FIQ

Table 3-9. Configuration Options (cont.)

Option	CPU Architecture	Valid values	Default
Vector_Catch (Cortex-M)	**Cortex-M**	**0x0 - 0x7ff**	**0x0**
The Vector_Catch option for Cortex-M processors specifies that selected exception vectors are to be watchpointed by the probe. See also Vector_Catch (ARM7, ARM9, ARM11) for behavioral differences with the ARM processors and Vector_Catch (Cortex-A) for the Cortex-A processors. When an instruction causing exception is committed for execution, execution halts due to a debug trap. The program counter points to the first instruction of the exception handler. The bit mask is defined as follows (decimal bit, hex field bit, and cause):			

Bit	Field	Cause
0	0x001	Reset
4	0x010	MMERR: MemManage exception
5	0x020	NOCPERR: UsageFault - coprocessor access error
6	0x040	CHKERR: UsageFault - checking error (alignment / div by 0)
7	0x080	STATERR: UsageFault - state info error (undefined inst / invalid EPSR.T or EPSR.IT)
8	0x100	BUSERR: BusFault
9	0x200	INTERR: Exception Entry or Return
10	0x400	HARDERR: HardFault

Special Probe Initialization (ARM and MIPS Only)

This section details the special probe initialization for ARM and MIPS.

MEP_JTAG_DIMENSION

In most cases the probe can automatically determine the JTAG configuration. It does this with a quick test when the JTAG interface is first initialized. In some cases, it might be necessary to declare the JTAG connection details by writing a JTAG descriptor to the MEP_JTAG_DIMENSION buffer. The format of the JTAG descriptor consists of the number of TAPs in the chain, followed by a sequence of numbers to specify how many TAPIR bits are in each TAP. When a descriptor is declared in this way, the probe disables the automatic detection process and uses the specified descriptor to manage the scan chain.

CntTAPs, CntIR1, CntIR2, ..., CntIRn

Example:

```
ew MEP_JTAG_DIMENSION = 2, 5, 7    // There are 2 TAPs, one has 5 IR bits,
                                      one has 7
```

MEP_JTAG_INIT0 and MEP_JTAG_INIT1

The MEP_JTAG_INITx descriptors can be used to define special JTAG initialization operations required by certain devices. The INIT0 descriptor, if defined, is scanned through right after the JTAG reset cycle completes, but before the standard JTAG initialization performed by Sourcery Probe. The second, if defined, is scanned through right after the standard JTAG initialization performed by the Sourcery Probe.

On most boards these can both be omitted. The Sourcery Probe initialization script template includes comments showing where these are typically used in the initialization process for those boards that require something special.

When used, the format is a list of 0-N *Scan Operation* frames:

```
ew MEP_JTAG_INIT0 = <scan-op> [ , <scan-op> ]  ...
ew MEP_JTAG_INIT1 = <scan-op> [ , <scan-op> ]  ...
ew MEP_JTAG_INIT0 = 0    // disable INIT0 descriptor
ew MEP_JTAG_INIT1 = 0    // disable INIT1 descriptor
```

Each *scan-op* frame is a free-form data buffer in the following format:

```
FF0SNCNT [ , SCANDATA ] [ , SCANDATA ]...
```

FF0SNCNT is a 32-bit word in big endian format, with eight flag bits in the MS bits and 20 bit scan count in the LS bits.

Table 3-10. MEP_JTAG_INIT0 and MEP_JTAG_INIT1 Field Descriptions

Field	Bit(s)	Description
IR/DR	0x80000000	IR if the MSB is a 1, DR if 0
P/U	0x08000000	End in PAUSE if 1, end in Update/Idle if 0
IN	0x01000000	Scan IN to TDI, ignore TDO
sncnt	0x000nnnnn	Scan bit count

SCANDATA is defined as an array of 32-bit words, where the LSB of the first word is shifted into TDI and out of TDO, then higher order bits from that word, then LSB of the next word, and so on. If the final word has less than 32 bits, those final bits are in the LS bit positions of the final word. The number of words of SCANDATA depends on the bit specified in the FF0SNCNT control word.

MEP_JTAG_SCAN and MEP_JTAG_SCAN_TAP

The MEP_JTAG_SCAN and MEP_JTAG_SCAN_TAP descriptors can be used to define special JTAG initialization operations required by certain devices. These descriptors are similar to the MEP_JTAG_INITx descriptors except that they perform their function immediately, unlike the MEP_JTAG_INITx descriptors that are preset in advance and are then used during JTAG initialization.

When used, the format is as follows:

```
ew MEP_JTAG_RESET = jrmode          // reset JTAG in mode 0,1,2

                                       0   Reset mode based on IJUT option
                                       1   Reset JTAG via TRST* signal
                                       2   Reset JTAG via 5xTMS=1 cycles

ew MEP_JTAG_UNRESET = jimode        // unreset JTAG and initialize in mode 0,1,2,4,8

                                       0   unreset, no initialization
                                       1   Enable multi-TAP support (see table below)
                                       2   Enable bypass test
                                       4   Enable multi-TAP auto detection
                                       8   Enable multi-TAP using MAJIC_JTAG_SCAN_DIM

       MT   MT_AUTO   MT_SDEF   Description
       --   -------   -------   -----------------------
       0      x         x     Whole chain mode (multi-TAP support is disabled)
       1      0         0     Multi-TAP support using MEP_JTAG_DIMENSION if defined,
                                 else auto detect
       1      0         1     Multi-TAP support using MEP_JTAG_DIMENSION if defined,
                                 else single TAP or whole chain mode
       1      1         x     Multi-TAP support, using automatic detection.

ew MEP_JTAG_SCAN = scan-list        // perform the specified list of scan operations on
                                       the default TAP; i.e., the TAP selected with the
                                       IJTS option.
ew MEP_JTAG_SCAN_TAP | (TAP << 4)= scan-list
                                    // perform the specified list of scan operations on
                                       the specific TAP
dw MEP_JTAG_SCAN                    // displays the results from last ew MEP_JTAG_SCAN
                                       or MEP_JTAG_SCAN_TAP
dw MEP_JTAG_SCAN_TAP                // displays the results from last ew MEP_JTAG_SCAN
                                       or MEP_JTAG_SCAN_TAP
```

Scan-List Format

Each *scan-list* frame is a free-form data buffer in the following format:

```
FF0SNCNT [ , SCANDATA ] [ , SCANDATA ]...
```

FF0SNCNT is a 32-bit word in big endian format, with eight flag bits in the MS bits shown in the table below and 20 bit scan count in the LS bits.

Table 3-11. Scan-List Field Descriptions

Field	Bit(s)	Description
IR/DR	0x80000000	IR if the MSB is a 1, DR if 0
P/U	0x08000000	End in PAUSE if 1, end in Update/Idle if 0

Table 3-11. Scan-List Field Descriptions

Field	Bit(s)	Description
IN	0x01000000	Scan IN to TDI, ignore TDO
OUT	0x02000000	Scan TDO out to probe, and recycle back into TDI
I/O	0x03000000	Scan into TDI and scan TDO out to probe
sncnt	0x000nnnnn	Scan bit count

Automatic Access Mode

In some target environments, if might be desirable or necessary for the Mentor Embedded Sourcery Probe to take some special action upon stopping and restarting execution, or periodically while stopped. For example, many targets have a watchdog timer that will reset the board if the software fails to access the watchdog timer periodically. Because the software stops executing when a breakpoint is hit, the watchdog timer will reset the board shortly after the first breakpoint. This reset is disruptive to the debug environment.

The simplest way to deal with an unexpected reset during debug is to disable the watchdog timer during development, and only enable it in the final production build of your software. However, because this is often impractical and sometimes impossible, the Mentor Embedded Sourcery Probe provides several features to automatically perform a user-defined set of memory access at key points in your debug session. Table 3-12 shows the list of accesses.

Table 3-12. User-defined List of Accesses

ON_RESET	Access is performed whenever the target is reset by the Mentor Embedded Sourcery Probe. This can be used to issue a wake up status indication in systems where the CPU is expected to identify itself after reset.
ON_STOP	Access is performed when program execution stops for any reason. In the case of breakpoints with pass counts that are managed by the Mentor Embedded Sourcery Probe, the access list is only performed when the pass count is reached. For breakpoints that include a conditional expression, then the ON_STOP accesses are performed each time the breakpoint is hit, even if the debugger automatically restarts execution.
ON_GO	Access is performed whenever the Mentor Embedded Sourcery Probe starts program execution.
IDLE_MODE	Access is performed repeatedly at the specified interval while program execution is halted. This is usually the best way to manage a watchdog timer that needs to be accessed periodically to prevent a reset or NMI.

Note

The ON_STOP and ON_GO accesses are also performed if execution pauses for a semi-hosting call or concurrent debug mode operation. Additionally, the IDLE_MODE accesses are enabled while paused for a semi-hosting call. However, because execution is automatically restarted as soon as possible when a concurrent debug operation requires execution to pause, the IDLE_MODE accesses do not take place while paused for a concurrent debug mode operation.

Note

The ON_STOP and ON_GO accesses are not performed when the Mentor Embedded Sourcery Probe performs instruction level stepping. They are performed if the probe or the debugger software performs stepping by setting a temporary breakpoint and running to it.

User Defined Access Lists

To use the ON_RESET, ON_STOP, ON_GO, and IDLE_MODE features, you must define a set of memory accesses to perform on those event(s). The Mentor Embedded Sourcery Probe provides various access list descriptors for defining the access sequence for each case. The *cmd_desc.maj* command file provides primitives for describing these access sequences. The *cmd_desc.maj* file should be read in the target initialization command file for your board if you intend to use any of these features.

Note

Do NOT change the *cmd_desc.maj* command file. It is read during initialization to define the primitives you require to define access sequences appropriate for your target.

The commands to preset these sequences are added to the target initialization file. Each descriptor is a maximum of 16 words long, in the following format:

```
ew MEP_ON_STOP_CMD   = en,  { @op, addr [ , data [ , mask ] ] }...
ew MEP_ON_GO_CMD     = en,  { @op, addr [ , data [ , mask ] ] }...
ew MEP_ON_RESET_CMD  = en,  { @op, addr [ , data [ , mask ] ] }...
ew MEP_IDLE_MODE_CMD = int, { @op, addr [ , data [ , mask ] ] }...
```

Where:

Table 3-13. Descriptor Components

en	enable: 0 to disable, 1 to enable
int	interval: 0 to disable, non-0 for interval in milliseconds. Although specified in milliseconds, timing accuracy is not maintained below 100ms granularity.

Table 3-13. Descriptor Components

op	opcode: $ucd_rd8 // 8-bit read operation $ucd_rd16 // 16-bit read operation $ucd_rd32 // 32-bit read operation $ucd_wr8 // 8-bit write operation $ucd_wr16 // 16-bit write operation $ucd_wr32 // 32-bit write operation $ucd_rmw8 // 8-bit RdModWr operation $ucd_rmw16 // 16-bit RdModWr operation $ucd_rmw32 // 32-bit RdModWr operation
addr	Address of the access (must be properly aligned for op size). If you are using an MMU, the address should be specified as a virtual address that is always accessible. The address is always 32-bits. On MIPS64 targets, the address is sign extended for kseg0/1 addresses.
data	Data value for $ucd_wr and $ucd_rmw (omit for $ucd_rd)
mask	Data mask for $ucd_rmw (1's are bits to replace with data)

Examples

The following examples demonstrate the use of the ON_RESET, ON_STOP, ON_GO, and IDLE_MODE features.

```
ew MEP_ON_RESET_CMD = 1, @$ucd_rd8, FFF00003
```

Defines an On-Reset command that reads the byte at 0xFFF00003 after the probe resets the target.

```
ew MEP_ON_STOP_CMD = 1, @$ucd_rmw16, 80000000, C00, F00
ew MEP_ON_GO_CMD   = 1, @$ucd_rmw16, 80000000, 300, F00
```

Defines On-Stop and On-Go commands that read a 16-bit value from 80000000, masks off bits 11..8, sets those bits to 1100 (On-Stop) or 0011 (On-Go), then writes the result back to 80000000. Other bits in that register are not changed by these commands.

```
ew MEP_IDLE_MODE_CMD = 0n250, @$ucd_wr32, 0x40000000, 0n5000
```

Defines an Idle-Mode command that writes the 32-bit value 5000 (decimal) to the register at 0x40000000 every 250 milliseconds.

```
ew MEP_ON_RESET_CMD  = 0
ew MEP_ON_STOP_CMD   = 0
ew MEP_ON_GO_CMD     = 0
ew MEP_IDLE_MODE_CMD = 0
```

Disables the On-Reset, On-Stop, On-Go, and Idle-Mode command descriptors.

Chapter 4
MON Command Language

This section provides details about the MON command language as it applies to the Sourcery Probes for ARM and MIPS.

The primary use of the MON command language within the GDB and Sourcery CodeBench environment is in Sourcery Probe target initialization (*.maj*) files. These initialization files configure the Sourcery Probe for a given target board and provide scripts for performing different reset and initialization functions.

You can also enter MON commands in the Probe Console view of the Sourcery CodeBench Debug perspective, or use them within gdb scripts to access Sourcery Probe features not directly available in GDB or Sourcery CodeBench.

The MON Command Line Debugger program accepts MON commands at the input prompt. (See Chapter 4 - MON Command Line Debugger for information on running and using the MON debugger.)

MON Command Basics

In learning the MON Command Language, it helps to know some of its design philosophy. Commands are usually formed from the first letter of each word in the spoken command name. Command names are of the form Action, Action Object, or Object Action (for example, Go, Display Word, or File Read).

Command names were chosen for being easy to remember, and the command mnemonics are simple abbreviations. The number of basic commands is kept to a minimum - there is just one Display command, for instance. Symbolic MON commands are not case sensitive.

MON Help

MON features a hierarchical help system. The following types of commands are available:

```
H         // shows the list of commands.
H D       // gives general help on display commands.
H DW      // gives detailed help on the display word command.
H ops     // shows the list of command operands for which on-line help
          // is available.
H addr    // gives detailed help on address operands.
```

Command Lists

You can enter multiple commands on a single line, separated by "**;**". For example, the following line sets the stack pointer and then displays it.

```
ew sp = a00ff000; dw sp
```

Command lines can contain C-style comments (for example, "/*This is a comment*/") and C++ comments (for example, "//...").

Comments are replaced with a single blank character when the command is being interpreted. Comments are useful for documenting command files.

MON Command Quick Reference

This section describes the available MON commands.

Table 4-1 shows the syntax of the available MON commands. For details on these commands see MON Command Details.

Note
You can use the H command for on-line help (or H H for help on using help).

Table 4-1. MON Command Quick Reference

Command	Syntax
+B (Enable Breakpoints), -B (Disable Breakpoints)	**Enable or Disable Breakpoints** **{+|-}B [*|** *{#number |addr}* **...]**
BC (Breakpoint Clear)	**Breakpoint Clear** BC [*|*addr*]
BL (Breakpoint List)	**Breakpoint List** BL
BS (Breakpoint Set)	**Breakpoint Set** BS [2] [addr [, [-]*number*]] ... ["{"*cmd_list*"}"]
BSH (Breakpoint Set Hardware)	**Hardware Breakpoint Set** BSH [*size*] [*asid*] arange [# *mask*] [= *vrange* [# *mask*]] [, [-]*number*] ["{"*cmd_list*"}"]
CF (Cache Flush), CFI (Cache Flush Invalidate), CI (Cache Invalidate)	**Cache Flush** **`C{F|I|FI} [I|D]`**

Table 4-1. MON Command Quick Reference (cont.)

Command	Syntax
DA (Display Alias)	**Display Aliases** **DA [*|***ident***]**
D (Display Data)	**Display or Find Byte** **DB[R] [***range***[,** *fmt***] [=***value* **[#** *mask***][,** *value* **[#** *mask***]] ...]]**
	Display or Find Double Word **DD[R] [***range***[,** *fmt***] [=***value* **[#** *mask***][,** *value* **[#** *mask***]] ...]]**
	Display or Find Half-Word **DH[R] [***range***[,** *fmt***] [=***value* **[#** *mask***][,** *value* **[#** *mask***]] ...]]**
	Display Quad Size Objects **DQ[R] [***range***[,** *fmt***]**
	Display or Find Word **DW[R] [***range***[,** *fmt***]** = *value***[#***mask***]] [**, *value***[#***mask***]]...]**
DC (Display Cache) - MIPS Only	**Display Cache Tags or Cache Memory**
DI (Display Information)	**Display Information** **DI**
DM (Display MMU Table / Translate Address) - ARM	**Display MMU Information / Translate Address** **DM [** *num [x]* **]** **DMX [V]** *addr*
DM (Display MMU Table / Translate Address) - MIPS and PowerPC	**Display MMU Information / Translate Address** **DM** **DMX** *addr*
DN (Display Names)	**Display Symbolic Name** **DN {*|***ident***|***ident****}**
DO (Display Options)	**Display Option** **DO[Q|V] [*|***pattern****|***cfg_option***]**
DS (Display Scripts)	**Display Scripts** **DS[Q|V] [***file_spec***]**
DV (Display Values)	**Display Values, Formatted** **DV** *string***[,** *expr***]**...
EA (Enter Alias)	**Enter Command Alias** **EA** *ident cmd_list*

Table 4-1. MON Command Quick Reference (cont.)

Command	Syntax
EB, ED, EH, EQ, EW (Enter or Fill Data)	**Enter or Fill Byte(s)** `EB[K] [range][,fmt] = value[, value] ...`
	Enter or Fill Double Word `ED[K] [range][,fmt] = value[, value] ...`
	Enter or Fill Half Word `EH[K] [range][,fmt] = value[, value] ...`
	Enter or Fill Word `EW[K] [range][,fmt] = value[, value] ...`
	Enter or Fill Quad Word `EQ[K] [range][,fmt] = value[, value] ...`
EN (Enter Names)	**Enter Symbolic Name** `EN ident = addr`
EO (Enter Option)	**Enter Option** `EO cfg_option = value`

FR (File Read)	**File Read**
	`FR C filename [p_value ...]`
	`FR M filename [addr]`
	`FR RD filename`

FW (File Write)	**File Write** **FW[A|O] {O|C|M}** {*file_name*|-|+} [*range*] **FW[A|O] {MDI}** {*file_name*} Description
G (Go, Go Interactive)	Starts processor execution G[I] [=`addr`] [`addr` ...] [{[`cmd_list`]}]
GOTO	**GoTo** (used in command files) `GOTO ident`
+H (Enable Hardware Names), -H (Disable Hardware Names) - MIPS Only	**Enable or Disable Hardware Names (for MIPS Only)** `{+H|-H}`
H (Help)	**Help** `H [command|op_key|CONTROL|OPS]`

Table 4-1. MON Command Quick Reference (cont.)

Command	Syntax
IF	**If** **IF** *expr* **{***then_cmds***} [{***else_cmds***}]**
KA (Kill Alias)	**Kill Aliases** **KA {*|***ident***}**
KN (Kill Names)	**Kill Symbolic Name** **KN {*|***ident***|***ident****}**
L (Load)	**Load Program** **L [[-[N]o {t|d|b|l|s}]***filename* **...]** **...**
LN (Load Names)	**Load Symbolic Names** **LN[A|O] [***filename***] ...**
MB, MD, MH, MW (Move, Move Reverse)	**Move** **M[R] [B|H|W|D]** *range***,** *addr*
MC (Memory Configuration) (ARM and MIPS Only)	**Memory Configuration** **MC [***range* **[, {{pwe|pwd}{dma|jam|inv}}] ...]**
MT (Memory Test) (ARM and MIPS Only)	**Memory Test** **MT** *range***[,{1|2|3|4|5|8|9}][,{H|V|Q|S}...]** **[,***repeat_cnt***]** **MT** *range***,8,***delay***[,{H|V|Q|S} ...]** **[,***repeat_cnt***]** **MT** *range***,{10|11|12}[,***data* **]** **[,***repeat_cnt***]**
+Q (Enable Quiet Mode), -Q (Disable Quiet Mode)	**Enable or Disable Quiet Mode** **{+Q|-Q}**

Table 4-1. MON Command Quick Reference (cont.)

Command	**Syntax**
R, RP, RT (Reset Processor)	**Reset Processor or Reset Target** **`R`** // Deprecated. Performs either RP or RT reset. See Ice_Reset_Output.
	Reset Processor **`RP`** Soft CPU-only reset.
	Reset Target **`RT`** Low-level hard reset of the target system. See Components of Initialization Files for recommended reset usage.
RETURN	**Return from command file or function** Only valid in scripts. Exits current script or function processing.
SHIFT/UNSHIFT	**Shift Command File Arguments** **SHIFT** [*number*] **UNSHIFT** [*number*|*]]
S (Step)	**Single Step, Step Forward, Step Over** **S[F|O][Q|V]** [=*addr*] [*number*] [{[*cmd_list*]}]
SP (Stop)	**Stop Processor Execution** SP Available only in concurrent execution mode
TC (Target Connect)	Makes a connection to the configured probe and target.
TD (Target Disconnect)	Disconnects from a connected target.
TS (Target Status)	**Displays current target connection status**

VL (Verify Load)	**Verify Load**
	`VL [[-[N]o {t|d|b|l|s}]` *`filename`* **`... ]`**
	`...`

W (Wait)	Wait for N milliseconds W [N]
! (Execute Operating System Shell)	Execute Operating System Shell ![*os_command*]

MON Command Details

The following pages explain each MON command. Each command listing contains its syntax, description, and examples.

+B (Enable Breakpoints), -B (Disable Breakpoints)

Availability: ALL

The Enable Breakpoint (+B) or Disable Breakpoint (-B) command enables or disables one or more software breakpoints.

Breakpoints must be previously set with the BS (Breakpoint Set) command. If no parameter is given, and there is a software breakpoint set at the current execute location, that breakpoint is enabled or disabled.

Usage

{+|-}B [*]

{+|-}B {*#number* |*addr*}...

Arguments

- *

  All software breakpoints are enabled or disabled.

- # number

  Specifies the number, as shown by the BL (Breakpoint List) command, of a software or hardware breakpoint to be enabled or disabled.

- addr

  Specifies the address of a software breakpoint to be enabled or disabled. addr must be an address in a valid code address space.

Examples

```
BS main First Second Third
-B *
+B main Third
```

BC (Breakpoint Clear)

Availability: ALL

The BC (Breakpoint Clear) command clears one or more software breakpoints.

Breakpoints must be previously set with the BS (Breakpoint Set) command. To temporarily disable a breakpoint, use the -B (Disable Breakpoint) command instead. If no parameter is given and there is a software breakpoint set at the current execute location, that breakpoint is cleared.

Usage

BC [*]

BC {*#number* |*addr*}

Arguments

- *

  All software breakpoints are cleared.

- # number

  Specifies the number of a software or hardware breakpoint to be cleared. See the BL (Breakpoint List) command.

- addr

  Specifies the address of a software breakpoint to be cleared. addr must be an address in a valid code address space. The addr syntax is shown in addr Address.

Examples

```
BC
BC *
BC my_sym
```

BL (Breakpoint List)

Availability: ALL, but only shows breakpoints set directly in MON.

The BL (Breakpoint List) command displays a list of all breakpoints currently set, showing the break address, initial pass count, remaining pass count, whether or not it is active (enabled), and the associated command list, if any.

If a non-empty command list is still active from a previous G (Go, Go Interactive) or S (Step) command, it will also be displayed.

Usage

BL

Arguments

BS (Breakpoint Set)

Availability: ALL, but usage in a MDI environment can be problematic as the higher level debugger is not aware of breakpoints set here.

The BS (Breakpoint Set) command sets one or more software breakpoints with optional pass counts and an optional command list.

Usage

BS[2] [*addr* [, [-]*number*]] ... ["{" *cmd_list* "}"]

Arguments

- addr

  Specifies the address where execution will break. This must be an address in a valid code address space. Note that if addr is not specified, the software breakpoint will be set at the current execute location.

- -

  The minus sign indicates the software breakpoint is temporary and will be removed when it is hit pass_count number of times. A temporary breakpoint is different from the non-sticky breaks that can be specified with the Go command which all disappear when the program stops for any reason.

- number

  This is by default decimal. See number for available number forms.

- pass_count

  Decimal number specifying the number of times the breakpoint location must be reached before the breakpoint is taken. At the time the command list is executed, and the pass counter is reloaded. If no pass_count is given, the breakpoint is taken every time the address is reached.

- cmd_list

  This is any valid command list. Specifies one or more debugger commands to be executed when program execution is stopped by this breakpoint. Curly braces surrounding the cmd_list are required. The command list may contain a Go or Step command, in which case the program will be resumed automatically. If present, the Go or Step command must be last. Combining IF and G commands in a breakpoint command list allows complex conditional breakpoints to be created. If no cmd_list is provided, execution simply stops.

Description

BS sets normal breakpoints and BS2 sets MIPS16/Thumb mode breakpoints. If no pass count is specified, a value of one is assumed. If a negative pass count is specified, the breakpoint is temporary: it will be removed automatically when it is taken. The program will be interrupted each time the break address is reached <number> times. At that time, the command list will be executed. There is no limit on the number of active breakpoints.

Examples

```
BS // set breakpoint at current PC.
BS 400:r // set breakpoint 400 (hex) bytes above reset vector
BS my_label // set breakpoint at my_label
BS @RA // set breakpoint at address in register RA.
BS my_label your_label
BS my_label {dv "\n executed my_label\n"; g}
BS 5020:0, 20 {dv "\nat 80005020 20 more times\n"}
BS my_label,-3 your_label, 2
BS func1, -5 {dv "\n at func1 fifth time\n"; dw var; dw (@ptr):d; g }
```

Note

The last example, like all debugger commands, must be entered on one line.

BSH (Breakpoint Set Hardware)

Availability: ALL, but usage in a MDI environment can be problematic as the higher level debugger is not aware of breakpoints set here.

The BSH (Breakpoint Set Hardware) command sets a hardware breakpoint with an optional pass count and an optional command list.

Usage

BSH *options* [*size*] [*asid*] *arange* [# *mask*]

[= *vrange* [# *mask*]] [, [-]*number*] ["{" *cmd_list* "}"]

Arguments

- *options* { I | R | W | S | A | T | B }...

  Flag word consisting of one or more of the following letters:

  I - Break on matching instruction fetch

  R - Break on Data Read

  W - Break on Data Write

  S - Size qualifier present, break only if access matches

  A - ASID qualifier present, break only if access matches

  T - Generate a Tracepoint signal without stopping.

  B - Break on Tracepoint (redundant unless T also given)

- *size* — { 8 | 16 | 32 | 64 }

  This is the access size. It is required if options has the value 'S', otherwise it is invalid.

- *asid*

  This is a number giving the ASID value. It is required if options has the value 'A', otherwise it is invalid.

- *arange* — [addr [addr] | addr L number]

  This is the memory address or address range which triggers the breakpoint. If the "addr L number" syntax is used, number is scaled by size if specified, otherwise it is taken as a byte count. This parameter is required, but a mask of 0 can be applied if the break should not be conditioned on the address (for example, fetching a certain value from any address).

- *vrange* — [value [value] | value L number]

  This is the value or range of values which will trigger the breakpoint when accessed. See expr Expression for valid value forms. If the "addr L number" syntax is used, number is taken as a byte count.

- *mask*

  Hexadecimal bitmask applied to the address or value range as part of the breakpoint condition. Zero bits in mask are “don't care” bits when testing the address or value.
- *number*

  This is by default decimal.
- *cmd_list*

  This is any valid command list, with G or S being last, if present.

Description

If no pass count is specified, a value of one is assumed. If a negative pass count is specified, the breakpoint is temporary: it will be removed automatically when it is taken. The program will be interrupted each time the break address is reached number times. At that time, the command list will be executed. The actual number of hardware breakpoints available and the options which can actually be used to condition them, is dependent on the target CPU's capabilities.

CF (Cache Flush), CFI (Cache Flush Invalidate), CI (Cache Invalidate)

Availability: ALL

The Cache commands are used to flush and/or invalidate the contents of the instruction and/or data caches. If an I or a D operand is specified, only the instruction (I) or data (D) cache is affected. If no operand is specified, both caches are affected.

Note

Support for CF (Cache Flush) and CI (Cache Invalidate) operations depends on features that are not provided by all processors. If the specific processor you are using does not support the operation, an error message is displayed.

Usage

CF [I|D]

CI [I|D]

CFI [I|D]

Arguments

- CF

  Flushes (writes back to memory).

- CI

  Invalidates without updating memory.

- CFI

  Flushes and then invalidates.

DA (Display Alias)

Availability: ALL

The DA (Display Alias) command shows the name and replacement text for one, or all currently defined aliases.

If the command is entered without a parameter, all aliases are displayed. Alias names are not case sensitive. See EA (Enter Alias) for more information about creating command aliases. See also Command Aliases.

Usage

DA [|ident]*

Arguments

- *

  Display all aliases. This is the default.

- *ident*

  The name of a command alias defined with the EA (Enter Alias) command. The ident argument is not case sensitive.

Related Topics

EA (Enter Alias)	Command Aliases

D (Display Data)

Availability: ALL

The D (Display) command displays the contents of the specified registers or memory locations in a variety of formats, or searches a memory range for the specified *value* list. The Find version of this command only supports target memory spaces. Registers and "MON local" address space might not be searched.

When a Display command is entered, it can be repeated for successive addresses by pressing the Enter key, until some other command is entered. The = *value* option causes successive Enter keys to repeat the search from where the last match left off. If no more matches are found, a message indicating that fact is displayed.

Usage

D[*type*]

D[*type*] *range* [, *fmt*]

D[*type*][R] *range* [, *fmt*][= *value* [# *mask*][, *value* [# *mask*]] ...]

Arguments

- *type*

  {B|H|W|D|Q} specifies the object size, where B is for Byte, H is for 16-bit Half-word, W is for 32-bit Word, and D is for 64-bit Double word. The default is the type specified in the previous Display command or W, if this is the first Display command.

Note

Find/Search on quad word objects (DQ command) is not supported.

- *R*

  The Reverse option causes the searching to start at the end of the range and scan backwards. (Note that type is then required if the DR "display registers" alias is active.)

- *range*

  This can indicate the location and number of objects to display or search. If range does not specify a start address, Display continues where the previous Display left off, or begins at virtual address 0 if it is the first Display command. See range Address Range.

- *fmt* - {d|u|o|x|X|f|e|E|g|G|c|s|i|I}

  The display and search value format; default is "X" (hex). For more information and to see a list of valid formats for each type, see fmt Data Format.

- *value*

  The data to match against and must be given in *fmt* type. For integer fmt types, value can be an expr. If fmt is "s", value is a string literal (string syntax is shown in string String Literal.).

If fmt is “i” or "I" then the MIPS mini-assembler is invoked to assemble the search value (see EB, ED, EH, EQ, EW (Enter or Fill Data).)

- *mask*

  A hex value that specifies which bits of *value* should be compared with memory. If mask is supplied only for the last value in the list, it will apply to all values in the list.

Examples

```
DW R0 R7             // Display the first eight general registers.
DB @my_str_ptr,s     // Display null-terminated string pointed
                     // to be my_str_ptr.
DW 2000:0 l 100,i = lui r0,0 #fc000000
                     // Find first MIPS LUI instruction in range.
                     // (Press <enter> to find successive LUIs)
DW 2000:1 l 6,i
a0002000:      08000803   j       0xa000200c
a0002004:      3c05bfc0   lui     a1,0xbfc0
a0002008:      00a00008   jr      a1
a000200c:      00000000   nop
a0002010:      010a0002   srl     zero,t2,0
a0002014:      014a0002   srl     zero,t2,0

DB _fdata _edata,s = “ERROR” #DF
                     // Search the specified range for “ERROR”, case
                     // insensitive (due to 0xDF mask value)
```

DC (Display Cache) - MIPS Only

Availability: ALL

The DC (Display Cache) script displays the cache tags and, optionally, the cache memory associated with the given address. Run this command file with no parameters to see a usage prompt.

Note

This is implemented as a script file. It is not a built-in MON command. See Command Script Files.

Usage

```
dc <mode> <address>

        <mode>    : IT = Instruction Tags
                  : IC = Instruction Cache and Tags
                  : DT = Data Tags
                  : DC = Data Cache and Tags
        <address> : Address of interest

NAME                 = VALUE         DESCRIPTION

  trgt_icache_linesize= 16           //Instruction/unified cache line size
  trgt_icache_memsize = 0x4000       //Primary instruction/unified cache
size
  trgt_icache_sets    = 4            //# of sets in instruction/unified
cache

  NAME                 = VALUE         DESCRIPTION

  trgt_dcache_linesize= 16           //Primary data cache line size
  trgt_dcache_memsize = 0x4000       //Primary data cache size
  trgt_dcache_sets    = 4            //Number of sets in data cache
```

Arguments

Examples

```
MON> dc IT 0
Address: 0x00000000
Tag:     0x00000000
Index:   0x00000000

Way 0 :  .ict0      00000000 (pa=0 v=0 l lrf)
Way 1 :  .ict256    00001000 (pa=4 v=0 l lrf)
Way 2 :  .ict512    00002000 (pa=8 v=0 l lrf)
Way 3 :  .ict768    00003000 (pa=c v=0 l lrf)
```

DI (Display Information)

Availability: ALL

The DI (Display Information) command shows the version numbers and connection information for MON or the MDI library and the Mentor Embedded Sourcery Probe.

Usage

DI

Arguments

DM (Display MMU Table / Translate Address) - ARM

Availability: ARM

Displays the current target virtual-physical memory map (MMU) setup in the CPU page descriptor tables or (DMX) translates the given virtual address to physical.

Usage

DM [*num [x]*]

DMX[V] *addr*

Arguments

- V

  Means Verbose mode and displays MMU table translation details.

- *addr*

  Is a virtual address in the default space.

- *num*

  Is a number: { 0-3 }

  0 - displays mixed table (same as no num provided case)

  1 - displays mixed and mapped tables

  2 - displays mixed and not mapped tables

  3 - displays mixed, mapped and not mapped tables

- *x*

  Is any argument and causes extra MMU table information to be displayed.

DM (Display MMU Table / Translate Address) - MIPS and PowerPC

Availability: MIPS

Displays the current target virtual-physical memory map (MMU) setup in the CPU page descriptor tables or (DMX) translates the given virtual address to physical. The built-in hard-mapped memory segments (MIPS kseg0/1) are also displayed.

Usage

DM

DMX *addr*

Arguments

- *addr*

  Is a virtual address in the default space.

DN (Display Names)

Availability: ALL

The DN (Display Names) command displays the values (addresses) of symbols known in the MON context.

Note

In order for MON to be aware of the program's namespace, you must use the LN (Load Names) command to load the symbolic names.

Usage

DN {*|*ident*|*ident**}

Arguments

- *

  Displays all names. This is the default. '*****' is a wildcard character and can be given alone or at the end of *ident*.

- *ident*

  Is a global datum, function, or user-defined symbol name (entered with Enter Symbolic Name EN ident = addr). *ident* is case sensitive. When used with '*', *ident* can be the starting characters of one or more global data, functions, or user-defined symbols.

DO (Display Options)

Availability: ALL

The DO (Display Options) command displays a table of specified configuration options with their current value and a brief description.

If the command is entered without parameters, all options are displayed. The Display Options Verbosely (DOV) command fully describes the option or options; it is best used with a specific option. The Display Options Quietly (DOQ) command displays only the option's name and current value. To modify configuration options, see the EO (Enter Option).

Usage

DO[Q|V] [*|*pattern**|*cfg_option*]

Arguments

- *

  Displays all options. This is the default. '*' is a wildcard character and can be given alone or at the end of *pattern*.

- *pattern**

  Displays all configuration options where the first letters match *pattern*.

- *cfg_option*

  Displays a specific option. The various options available are described in Configuration Options.

DS (Display Scripts)

Availability: ALL

The DS (Display Scripts) command displays the name and call entry points of any MEP script files found on the current search path. The default script files extension is *.maj* and if no file extension is given then only *.maj* files are shown, otherwise only files matching the given extension are shown.

Usage

DS[Q|V] [*file_spec*]

Arguments

- Q

  Means Quiet mode.

- V

  Means Verbose mode. In verbose mode, each directory searched is identified/displayed.

- *file_spec*

  Has the form : { A-Z | a-z | 0-9 | _ | . | * }...

Related Topics

File Search Order

DV (Display Values)

Availability: ALL

The DV (Display Values) command allows you to generate formatted output. The format string controls the operation of DV (Display Values) much like the format string in a C printf() statement.

Usage

DV *string*[, *expr*] ...

Arguments

- *string*

  A string literal containing characters to be displayed and conversion specifiers. For each conversion format specification, a corresponding expr Expression argument is required to provide the value to be displayed. (The *string* syntax is shown in string String Literal.)

- *expr*

  All the conversion specifiers defined for the “C” printf() function are supported, except the use of ***** to specify dynamic field width or precision. Also, as an extension, the conversion letter can be preceded by an object size specifier: **B**, **H**, or **W** (or **L**) indicating the value to be displayed is a Byte, Half-word, or Word sized object located at address *expr*.

  The size and conversion letters combine to determine whether to display the value of *expr* or the data stored at the address *expr*. For formats **s**, **e**, **f**, and **g**, the data at the address *expr* is always displayed. For all other formats, the value of *expr* is displayed, unless a size specifier is used (in which case the data at *expr* is displayed, as demonstrated in the second example below).

Examples

```
DV "Hello, world!\n"
```

```
DV "Byte at %x is %02bx\n", global_char_var, global_char_var
```

EA (Enter Alias)

Availability: ALL

The EA (Enter Alias) command creates an alias (synonym) for a list of one or more commands. The alias can then be used as a command name. EA (Enter Alias) is normally used to create a short abbreviation (one or more characters) for a longer command or sequence of commands that are frequently used.

Usage

EA *ident cmd_list*

Arguments

- *ident*

  The name of the alias that is being created or re-defined. When *ident* is used as an alias name, *ident* is not case sensitive. (See ident Identifier.)

- *cmd_list* - *command* **[**;*command*]**...**

  One or more MON commands, separated by semicolons. If the last command in *cmd_list* is not complete (missing some parameters at the end) the missing parameters must be provided when the alias is used.

Description

When a MON command is being processed, the MON first checks to see if the command name matches an existing alias name, ignoring alphabetic case. If an alias match is found, the alias name is replaced by the text of *cmd_list*, and the command is re-scanned. If an alias match is not found, the MON checks for built-in commands. This means that aliases can be used to re-define existing built-in commands, and the alias replacement text can contain other alias names. Recursive alias references are not supported, however.

If *cmd_list* includes a "FR (File Read) C file" command, the command file will be read in Quiet mode to provide the illusion that the alias name is a built-in command.

Aliases can be displayed with the DA (Display Alias) command, and removed with the KA (Kill Alias) command.

Examples

```
EA DIA DW @pc l 10,i // Disassemble instructions from current pc
ea ms fr c my_script // Run the command file named my_script.maj
ea rc fr c           // Read command file without echo. Note that since
                     // the command file name is not specified, it must
                     // be provided when the rc alias is subsequently
                     // used.
```

EB, ED, EH, EQ, EW (Enter or Fill Data)

Availability: ALL

The Enter command allows the contents of the specified registers or memory locations to be altered. If a value list is not provided, the Enter command displays the current value of each object in turn, prompting for a new value (interactive mode). If a range is not supplied, the Enter command picks up where the last Enter command left off. EW with i or I fmt invokes the MIPS mini-assembler.

Note

Only hex input values are supported for quad word writes (EQ command). A quad word value may be separated '.' at the double word value.

Usage

E[*type*][K] [*range*][,*fmt*]

E[*type*] [*range*][,*fmt*] = *value*[, *value*] ...

E[*type*] [K] [*range*][,*fmt*] = *value*[, *value*] ...

Arguments

- *type*

  Specifies the object size. Has the form: {**B**|**H**|**W**|**D**|**Q**}. where **B** is for Byte, **H** is for 16-bit Half-word, **W** is for 32-bit Word, **D** is for 64-bit Double word and **Q** is for 128-bit Quad word. The default is the *type* specified in the previous Enter command or **W**, if this is the first Enter command.

- *K*

  Causes input to be read directly from the keyboard, even when reading commands from a command file. It is ignored if "= *value*" is given. If **K** is not specified and an Enter command without a *value* list is executed from a command file, the input data items are also read from subsequent lines of the command file.

- *range*

- Specifies the location(s) where the values will be stored. The syntax of *range* is described in range Address Range.

  If *range* does not specify a start address, Enter continues where the previous Enter left off, or begins at virtual address **0** if it is the first Enter command. If *range* includes a length or end address, an interactive Enter command will automatically terminate after the last value is entered. If *range* does not specify a number of objects, the default range is determined by the number of values supplied. And if no values are supplied, the default *range* is unlimited.

- *fmt*

  Specifies the display and data entry format. Has the form: {**d**|**u**|**o**|**x**|**X**|**f**|**e**|**E**|**g**|**G**|**c**|**s** |**i**|}
  The default format is **X** (hex). The **i** and **I** (instruction) format is supported only for MIPS

processors, and invokes the mini-assembler. For more information and to see a list of valid formats for each *type*, see fmt Data Format.

- *value*

  Is the data to be written to the location(s) specified in *range* and must be given in *fmt* type. For integer *fmt* types, *value* can be an *expr*. If *fmt* is "**s**", *value* is a *string* literal. If *fmt* is "**i**", *value* is an instruction. For more information on *fmt* and *expr*, see fmt Data Format and expr Expression, respectively.

  If an explicit *value* list is not supplied, Enter displays the location and current value of each object in turn, prompting for a new value interactively. Entering a backslash "\" instead of a value will cause the Enter command to re-display the previous location and its current value for modification or verification. Hitting the <enter> key by itself will skip over the current object without modification. A period "." instead of a value will terminate the interactive Enter command.

Description

If a value list is supplied, the Enter command stores the values immediately without reading the old values (that is, there is no interactive prompting). If the number of values supplied would overflow the specified range, the excess values are ignored. If the supplied values do not completely fill the range, the value list will be replicated as necessary to fill the range.

Note

Such replicating "fill" operations are implemented as an Enter followed by a "destructive" overlapping Move. When the entire range is read/write memory, the effect is as expected. But the results might not be intuitive if write-only or read-only resources are involved (for example, on MIPS processors, **ew r0 r31=123** will have the effect of filling the registers with 0 because r0 is architecturally defined as being 0 and that propagates to the other registers in the overlapping move).

Examples

```
MON> eb r2=55               // Sets R2 to 0x55.
MON> eb 0:1,c =a,,, ,b,,, ,c         // the string "a, b, c"
MON> eb 0:1 l 10,d = 16,17,18        // fill with value pattern
MON> ew $saved=@loop        // Saves the contents of the variable
                            // loop in the MON local variable
                            // $saved.
MON> ew $saved=@$saved+1    // Increments the contents of the
                            // MON local variable $saved.
MON> ew 1028:0,i = mfc0 a0,$12       // use MIPS mini-assembler
```

EN (Enter Names)

Availability: ALL

The EN (Enter Name) command is used to create user-defined names within the MON. The name is entered in MON's internal symbol table with the specified address as its value. The name can then be used in place of the address in all MON commands.

Usage

EN *ident* = *addr*

Arguments

- *ident*

  The symbol name to be created. *ident* is case sensitive.

- *addr*

  The address to associate with the symbol name. See addr Address.

Examples

```
EN flash_cfg = fffffc00  // name for flash memory configuration block
```

Related Topics

See Register Definition File for defining names, bit fields, and attributes to registers and MON local variables.

EO (Enter Option)

Availability: ALL

The EO (Enter Option) command provides a mechanism to configure the operation of the Mentor Embedded Sourcery Probe.

The various configuration options available and their valid values are described in Configuration Options.

Note
For a list of all available configuration options with brief descriptions, use the DO command. To see a full description of a configuration option, use DOV *cfg_option*.

Usage

EO *cfg_option* = *value*

Arguments

- *cfg_option*

  The long or short name of a valid MON configuration option.

- *value*

  Depends on the specific option.

Examples

```
eo ice_jtag_clock_freq = 25  // sets the JTAG clock to 25MHz

eo ijcf = 25   // does the same
```

FR (File Read)

Availability: ALL

This command opens a file for reading. The type of file is specified by the first operand.

Note

Command files can contain FR C commands which execute other command files. Command file reads can be nested up to 20 levels deep.

Usage

FR C *file_name* [*p_value* ...]

FR C *label* [*p_value*]....

FR C *file_name*[.*label*] [*p_value*]....

FR M *file_name* [*addr*]

FR {RD} *file_name*

Arguments

- **M**

  Reads a memory image file (binary or Motorola s-record file). The default filename extension is *.mem*.

- **RD**

  Reads user-defined register names from a Register Definition File. See Register Definition File for details. The default filename extension is *.rd*.

- *file_name*

  Is the name of the file to be read. If *file_name* does not include an extension, the MON supplies the default file name extension. If no extension is desired, *file_name* should end with a "**.**", which will be removed before opening the file. MON searches for *file_name* using the algorithm as described in File Search Order.

- *addr*

  The starting address where the memory image will be loaded. It is not necessary that this address match the starting address used to write the file. *addr* is optional on memory image files containing Motorola S-Records. If given, *addr* specifies an offset relative to the addresses in the S-Record file.

- **C**

  Reads commands from the specified file. The default filename extension is *.maj*, but if no match is found then *.cmd* is tried next (for backward compatibility). For details on how to create and use command files see Command Script Files. Also, see SHIFT/UNSHIFT.

  This subcommand causes the debug monitor to read commands from the specified file or position within a file identified by a label. If only a label is supplied then the current file is assumed. When all the commands in the file have been processed or a return command has been encountered, the monitor resumes reading commands from its previous input source, ultimately returning control back to the console. This is an easy way to input a standard set of commands or to quickly recreate an earlier session. The file can be created manually or can be the result of the File Write (FW C) command. If the FR C command is part of a multi-command line, the remaining commands on the line will be executed after all commands in the new file have been executed.

- *label*

  A function label within the referenced file_name. Function labels differ from normal goto labels by the use of two ::'s at the start of the label.

Note

Referencing labels allows for a function call like processing and a label has precedence over a file_name in this context. Also, a label function call can be done without using "FR C" as a direct command, but this method only allows references to labels that have been identified as function labels by the use of two :'s in front of the label declaration. Command files can be nested up to 20 levels deep.

- *p_value*

  A string that will be substituted for a parameter in the command file. A parameter is of the form: *$$n*, with 1 <= n <= 99.

FW (File Write)

Availability: ALL

The FW (File Write) command allows you to write a file. The type of file is specified by the first parameter.

Usage

FW[A|O] {O|C|M} {file_name|-|+} [range]

FW[A|O] {MDI} {file_name}

Arguments

- [**A**|**O**]

  Means append to or overwrite the indicated file unconditionally. Normally, if the file specified exists, you will be prompted for permission to overwrite or (for command and output files) append to it. To avoid that prompt, use the **FWA** command to append without prompt or the **FWO** command to overwrite without prompt.

- **M**

  Specifies a memory image file (binary). The default filename extension is *.mem*.

- **C**

  Specifies a command file; each command subsequently entered is written to the command file before it is executed. This makes it easy to capture a command sequence which can be replayed later with an FR command. The default filename extension is *.maj*.

- **O**

  Specifies an output capture log file. The default filename extension is *.out*.

- **MDI**

  Specifies a MDI Configuration file. The default filename extension is *.cfg*.

- *file_name*

  Is the pathname of the file to be written, relative to the current working directory. If *file_name* does not include an extension, MON will supply the default file name extension. If no extension is desired, *file_name* should end with a "**.**", which will be removed before opening the file.

- *range*

  Specifies the region of memory (or block of registers) to be written to the memory image file. See range Address Range. *range* and *addr* specify the range of data to be written to a Memory image file.

- {+|-}

  Means to suspend/restart writing to a file (O and C only). Once writing to a Command or Output file has been initiated, output can be temporarily suspended with "**-**" and later resumed with "+".

- *p_value*

  A parameter value (Valid only with “FR C”).

Description

Doing a File Write to the **O** (Output) file type causes each line in the MAJIC® tool view (including the echo of commands entered) to be logged into the specified file. Doing a File Write to a **C** (Command) file type records only the commands entered, but not the prompts and responses from MON. This is a convenient way to create script files that can be used to automate repetitive command sequences or to quickly recreate an interrupted debugging session.

Memory image files contain raw binary data “uploaded” from the target. They normally represent the contents of some range of memory at the time they were created, but they can also contain a dump of the processor’s register contents. The file contains no control information (such as the original address *range* written to the file), so a Memory file can be written from one location and later read back into a different location.

G (Go, Go Interactive)

Availability: MON, or a script file under MDI

The G (Go, Go Interactive) command starts or resumes execution of the program.

Execution continues until a breakpoint or the end of the program is encountered. The *I* option starts execution in interactive mode. This mode allows a subset of debugger commands to be used while the target is executing. The SP (Stop) command interrupts the running program and returns MON to normal debug mode. Note that MON‘s interactive mode command prompt differs from the normal prompt (for example, MON(r) means the target is running).

Note

You must use this command from a command script in MDI and the current execute state must not change. For example, the script can execute code and hit a breakpoint, but the current execute state should be the same as it was when the script started running.

Usage

G[I] [=*addr*] [*addr* ...] [{[*cmd_list*]}]

Arguments

- =*addr*
- If specified, execution begins at the addr address. Otherwise execution begins at the current Program Counter location.

Note

Some programs require initialized data sections or registers to be reloaded before the program may be restarted from its entry point. In such cases, the **Load** command (L (Load)) should be used rather than **G** =*addr*.

- *addr...*
- Remaining addresses on the command line specify temporary (non-sticky) breakpoints, which will disappear when execution stops for any reason.

Note

Each addr must be a valid code address.

- *cmd_list*
- If specified, the list of commands will be executed each time execution stops for any reason. This type of automatic command list is useful for displaying interesting values every time execution stops. If execution stops due to hitting a breakpoint that also has an associated command list, the breakpoint‘s command list is executed first. A *cmd_list* can contain a Go or Step command, in which case program execution will resume automatically. Note that it is legal for additional commands to follow a Go or Step command in a command list. They will be stacked for execution in the proper order when execution finally stops for the last

time, but this can be confusing. Ensure that a Go or Step is the last command to be executed in the list to avoid this situation.

Curly braces surrounding *cmd_list* are required and the entire Go command must be entered without any carriage returns. The current "end execution" *cmd_list* can be displayed with the BL command.

Note

It remains in effect until canceled by a Go or Step command with a new or empty command list (for example, **G {}**). Performing target access via cmd_list's is driven from the host debugger. Sourcery Probes also provide a method for the probe itself to perform a sequence of memory accesses. See Automatic Access Mode.

GOTO

Availability: ALL

The GOTO command is used to change the order of command execution when reading commands from a command file. It causes the command file reader to jump to the line following the specified label.

Usage

:*label*

::func_label

GOTO *label*

Parameters

- label or func_label

  Any valid identifier string. label is a parameter of type ident Identifier. *ident is* the name of the label being defined or referenced. *ident* is not case sensitive.

Description

Labels are defined in the command file by a line of the form:

```
:label
::func_label
```

The colon does not have to be in the first column of the line, but there must be no white space between the colon and the label. Two colons (::) identify the label as a function call entry point, but it is also valid to branch to such a label with the GOTO command. Functions are normally called using a direct reference to *func_label* as a command or using the fr c command. See H (Help).

GOTO commands can precede or follow the corresponding label definition. Label definitions and GOTO commands have no effect when reading commands from the console, but they will be saved in a command output file if command logging is in effect.

The GOTO command, in combination with MON local variables and the **IF** command, can be used to construct arbitrary conditional blocks and loops in command files.

Examples

```
ew $strptr = my_str_var   // set $strptr to address of my_str_var
ew $strlen = 0   // set $strlen to 0
:LOOP
eb $char = @.1$strptr   // read next character
if (@.1$char == 0) {goto DONE}   // if 0 then exit the loop
ew $strlen = @$strlen + 1   // else increment $strlen
ew $strptr = @$strptr + 1   // and advance $strptr
if (@$strlen < 100) {goto LOOP}   // then loop back
dv "ERROR: $strlen overflow\n"
goto EXIT
:DONE
```

```
dv "$strlen = %d\n", $strlen
:EXIT
```

+H (Enable Hardware Names), -H (Disable Hardware Names) - MIPS Only

Availability: ALL

This command enables/disables the display of hardware register names rather than software names.

Usage

{+|-}H

Arguments

H (Help)

Availability: ALL

The H (Help) command displays general or specific information about MON commands and operands. If no parameter is supplied, a brief summary of each command is displayed.

Usage

H

H *command*

H OPS

H *op_key*

H CONTROL

Arguments

- *command*

  The name of a MON command. The syntax and description of that command is displayed, along with basic information about its operands.

- **OPS**

  A summary of the MON monitor operands is displayed showing a list of operands for which help screens exist, along with their *op_keys.*

- *op_key*

  The syntax and description of the specified operand is displayed.

- **CONTROL**

  A list of flow control features for command files is displayed.

IF

Availability: ALL

The **IF** command supports conditional execution of MON commands. It is useful in command files, where the GOTO command can be used to conditionally alter the flow of control.

Note

Like all MON commands, the entire IF command must be entered on one line.

Usage

IF *expr* **{***then_cmds***}** **[{***else_cmds***}]**

Arguments

- *expr*

  An address expression. It is evaluated and if the resulting value is non-zero, the commands in the *then_cmds* are executed. If the value is zero and the *else_cmds* is specified, those commands are executed. Remember that a symbol evaluates to its address unless preceded by an @ operator.

  ```
  then_cmds - cmd_list
  else_cmds - cmd_list
  ```

Note

Curly braces must surround the *cmd_list*, and the entire **IF** command must be entered on one line.

Examples

```
ew $count = 0
:LOOP
if (@$count < 10) {/*then*/ ew $count = @$count+1} {/*else*/ goto DONE}

if (@uart_creg & 0x80) == 0) {goto LOOP}
:DONE
```

The example uses the MON variable $count to control the iterations through a loop that is waiting for a certain UART control register to be cleared.

KA (Kill Alias)

Availability: ALL

The KA (Kill Alias) command deletes the name and replacement text for one, or all of the currently defined command aliases.

Command aliases are created with the Enter Alias command. See EA (Enter Alias) for more information.

Usage

KA *

KA *ident*

Arguments

- *

  Remove all aliases.
- *ident*

  The name of a specific alias to remove. *ident* is not case sensitive.

Related Topics

DA (Display Alias)	EA (Enter Alias)

KN (Kill Names)

Availability: ALL

The Kill Names command deletes one or more symbols from MON's symbol table.

Usage

KN {*|*ident*|*ident}**

Arguments

- *****

  Kills all symbols. '*****' is a wildcard character and can be given alone or at the end of *ident*. When used with *ident* all matching names are deleted.

- *ident*

  Is a global data, function, or user-defined (entered with EN (Enter Names)) symbol name. *ident* is case sensitive. When used with '*****', *ident* can be the starting characters of one or more global data, functions, or user-defined symbols.

Related Topics

EN (Enter Names) LN (Load Names)

L (Load)

Availability: ALL

The L (Load) command downloads one or more executable files to target memory, loads symbol information from the files into the MON's symbol table, and prepares the program for execution. The -[**N**]**O** *scn_types* operands can be used to specify the section types to load or execute from loading for subsequent files.

Note

In most cases the program should be loaded via Sourcery CodeBench instead of using the MON load command.

Usage

L [[-[N]O *scn_types*] *filename* ...] ...

Arguments

- *scn_types*

  A set of letters specifying section types to load (-O) or not load (-NO):

t	text (program code)
d	data (initialized data)
b	bss (uninitialized data)
l	literals (read-only initialized data)
s	symbols

- *filename*

  Specifies an executable file to be loaded according to the current *scn_types*.

Description

In addition to loading the program files and symbol tables, the Load command can produce a number of other "side effects" controlled by several configuration options (see Configuration Options). Specifically:

- If the Reset_At_Load option is on (the default), Load first performs a Reset operation equivalent to the R command (see R, RP, RT (Reset Processor)).
- If the Load_Entry_Pc option is on (the default), after the load is completed, the PC is set to the entry point address contained in the first *filename*.

Each of the file names and their respective *scn_types* are remembered for future Load commands until explicitly changed in a subsequent Load command.

So if there is no *filename* explicitly specified, the current program is reloaded. In this case the symbol table is not normally reloaded, but you can use the -O option to force a reload. If new files to be loaded are specified, the existing global symbol table is purged and symbols are loaded from the new files by default.

Examples

```
L               // reload the current program (using the current options)
L -o db         // reload only data and bss sections of current program
L -o ts myprog // load text and symbols of myprog
L myboot -no s myprog1 -o tdbls mymain
                // load all sections and symbols of myboot, load all but
                // symbols of myprog1, load all of mymain
```

Related Topics

VL (Verify Load)

LN (Load Names)

Availability: ALL

The LN (Load Names) command reloads symbols for the current program into MON’s symbol table, or loads symbols from the specified files. The new symbols will replace any existing symbols by default, or they can be added to the existing symbols.

Usage

LN[A|O] [*filename*] ...

Arguments

- **LN, LNO**

  Overwrite the existing symbol table.

- **LNA**

  Add to the existing symbol table.

- *filename*

  The name of an executable file whose symbols are to be loaded.

MB, MD, MH, MW (Move, Move Reverse)

Availability: ALL

This command copies data in `range` to the destination beginning at *addr*.

The source data and destination address do not need to be in the same address space. For example, registers can be dumped to or loaded from memory by the Move command.

Usage

M[*type*] *range*, *addr*

MR[*type*] *range*, *addr*

Arguments

- *type*

  Specifies the object size. Has the form: {**B**|**H**|**W**|**D**}. where **B** is for Byte, **H** is for 16-bit Half-word, **W** is for 32-bit Word, and **D** is for 64-bit Double word. The default is the *type* specified in the previous Move command or **W**, if this is the first Move command.

- *range*

  Specifies the address space plus starting and ending addresses of the source data. See range Address Range for more details.

- *addr*

  Specifies the address space plus the starting address of the destination. The ending address is implied by the length of *range*.

Description

The data is normally copied forward from the starting addresses in the source and destination ranges, one *type*-sized piece at a time, with the resultant predictable destructive effect if a portion of the range implied by *addr* falls within *range*. If **R** is specified, the command becomes Move Reverse and the data will be copied backwards from the ending address in the source and destination ranges. In this case an overlapping upward move will be non-destructive, while an overlapping downward move will be destructive. Of course if the source and destination are in different address spaces, a move can only be destructive if the two spaces overlap in the same physical memory.

Examples

```
mw a0 a3, t0              // move argument registers a0..a3 to t0..t3.
mw a1000000 l 4, r1       // move four words from memory to registers
                         // r1..r4.
mrb 1000:1 10fe:1,1001:1 // move up one byte non-destructively.
```

MC (Memory Configuration)

Availability: MIPS and ARM

The MC (Memory Configuration) command allows display and configuration of the following properties of the target's physical address space:

- Invalid address ranges.
- Address ranges where the Sourcery Probe can use the DMA access method to read/write values, as opposed to instruction jamming.
- Address ranges where partial word access is allowed.

Usage

MC [*range*[, *mc_opt*] ...]

Arguments

- *range*

  Specifies a range of physical memory to be displayed or where the specified options apply. *range* must specify the physical address space (**:p**). See Table 4-7.

- *mc_opt* - {**JAM | DMA | INV**}

  {**PWD | PWE**}

  { DW = [8 | 16 | 32] }

  { **RO**|**RW** }

  JAM - The Sourcery Probe uses "instruction jamming" (executing Load/Store instructions) to access target memory.

  DMA - When supported by the target processor, the Sourcery Probe uses DMA to access target memory.

  INV - The Sourcery Probe never accesses the target memory.

  PWD - Partial word access is disabled, so the Sourcery Probe reads/modifies/writes full words when accessing the target memory.

  PWE - Partial word access is enabled, so the Sourcery Probe accesses bytes and half words in a single operation.

  DW - Bus/Data width.

  RO - Read only memory range.

  RW - Read/write memory range.

Description

Entered without any parameters, MC displays all available memory configuration information. If range is given but not *mc_opt*, the configuration of the specified address range is displayed. If an *mc_opt* is given, the setting is applied to the specified address range.

Related Topics

MC (Memory Configuration) (ARM and MIPS Only)

MT (Memory Test)

Availability: MIPS and ARM

The MT (Memory Test) command initiates a test of the target’s memory system, or one of three “scope loops”.

Usage

MT *range* [,*mt_id*][,{H|V|Q|S} ...] [,*repeat_cnt*]

MT *range*,8,*delay*[,{H|V|Q|S} ...] [,*repeat_cnt*]

MT *range*,*mt_loop*[,*data*] [,*repeat_cnt*]

Arguments

- *range*

  Specifies memory space, starting and ending addresses to be tested.

- *mt_id*

  Is a decimal number specifying the test type. The default is 9.

1	Basic patterns
2	Walking 1’s and 0’s
3	Rotating address
4	Inverted rotating address
5	Partial word access
8	Refresh
9	Each of 1, 2, 3, 4, and 5 in turn.

- *mt_loop*

  Is a decimal number specifying a type of scope loop.

10	read only
11	write only
12	write then read

- *delay*

  Is a decimal number specifying the delay time in milliseconds, between writes and reads. It applies only to test 8 (refresh test).

- *data*

  Specifies that *data* and its one’s complement are alternatively written. (Required if *mt_loop* = 11 or 12.)

- **H**

  Halt-on-error: prompt to abort testing upon error.

- **V**

  Verbose: constant updates on test in progress.

- **Q**

  Quiet: pass completions are not reported individually.

- **S**

  Silent: errors are not reported individually.

- *repeat_cnt*

  Is a decimal number specifying the number of times to run the test. Default is forever. Either *mt_id*, *mt_loop*, **H**, **V**, **Q**, or **S** is required if *repeat_cnt* is to be specified.

+Q (Enable Quiet Mode), -Q (Disable Quiet Mode)

Availability: ALL

This command enables or disables Quiet mode of command file playback. Normally, MON prompts and commands read from the command file are displayed just as they would be if the commands were entered from the keyboard. But, when Quiet mode is active, MON does not display prompts and commands while reading commands from a file.

Note

Quiet mode is automatically turned on while a command alias is being executed. When the alias command is finished, the original state of Quiet mode is restored.

Usage

{+Q|-Q}

Arguments

R, RP, RT (Reset Processor)

Availability: ALL

The RP (Reset Processor) command does a soft reset of the processor only. In many cases the reset is simulated by setting certain registers to the values they would have after a reset.

Usage

R

RP

RT

Description

The RT (Reset Target) command resets the whole target system via the system reset pin on the debug connector. This feature requires that the target board have the system reset pin of the debug connector wired to the board's reset circuit.

The R (Reset) command performs either an RP or RT, depending on how the Ice_Reset_Output option in Table 3-9 is set.

Certain CPU registers will be initialized as specified in the processor's data sheet for a reset operation, and the current execute location (PC) will be set back to the reset vector, unless overidden by the Reset_Address option.

Note

Most Mentor Embedded Sourcery Probe Target Initialization scripts define RTNI and RTI aliases for managing target resets. Those use an RT type reset with additional operations before and after the reset so its impact can be managed.

Arguments

Related Topics

Components of Initialization Files

RETURN

Availability: ALL

The RETURN command exits processing of the current script or function call. It is not a valid command except within the scope of a script.

Usage

RETURN

Arguments

SHIFT/UNSHIFT

Availability: ALL

The SHIFT and UNSHIFT commands change the correspondence between the arguments supplied on an “**FR C** *filename*” command and the formal parameter tokens within the command file.

Usage

SHIFT [*number*]

UNSHIFT [*number*|*]

Arguments

- *number*

  The number of arguments to shift or unshift

- *

  Only valid for UNSHIFT, and it restores the arguments so that **$$1** again refers to the first argument.

Description

Normally, the first argument is substituted for $$1, the second argument for $$2, and so forth. The SHIFT command increments the argument number that corresponds to each parameter number, effectively shifting the argument array so that a given range of parameter numbers refer to a higher range of arguments. The UNSHIFT command reverses this effect. Argument shifting is very useful when you want to perform the same series of actions repetitively on an unknown number of arguments (or groups of arguments).

Note

The $$0 parameter is also affected by shifting: if there were originally 10 arguments, after a SHIFT 2 command $$0 will be replaced with 8. $$* is not affected by shifting - it is always replaced with the entire argument list.

Examples

The following command file will display the contents of a series of ranges. It expects an argument list of the form:

```
   addr count [addr count] ...
if ($$0 < 2) { dv "Expected address and count\n";
               goto done }
:loop
   dw $$1 L $$2
   shift 2
   if ($$0 >= 2) { goto loop }
:done
```

Note

The first if command in the example, like all MON commands, must be entered on one line.

S (Step)

Availability: MON, or a script file under MDI

This command executes number instructions (default is 1) starting at addr (default is the current execute address), one at a time. Execution terminates after number instructions, or when a breakpoint is encountered. If a command list is given, it is executed every time execution or stepping stops. It remains in effect until canceled by a Step or Go command with an empty command list (for example, **S {}**). A Step command can be repeated (except for the effect of =*addr*) until some other command is entered, by pressing `<Enter>` at the MON> prompt.

Usage

S[F|O][Q|V] [=*addr*] [*number*] [{[*cmd_list*]}]

Arguments

- O

  Step Over (SO) allows the current instruction to fully complete before returning control to the debugger prompt. In this mode, both subroutines and exceptions are allowed to finish before control is returned to the user.

- F

  Step Forward (SF) steps into calls, but over exceptions (including interrupts).

- V

  Verbose mode. When a step count is given, each instruction will be displayed before it is executed. A side effect of this is that breakpoints are not enabled.

- Q

  Quiet mode. When a step count is given, nothing is displayed until execution terminates, at which point the next instruction to be executed is displayed.

- Q/V

  If neither Q nor V are specified, the mode of the previous Step command is retained, with Quiet mode as the initial default.

- =*addr*

  - If specified, this is the address where stepping begins. Otherwise stepping begins at the current Program Counter location.

Note

The address must be in a valid address space for instructions. addr must be an address in a valid code address space.

- *number*

  The (decimal) number of instructions to be executed. If not specified, one (1) instruction will be executed. In Quiet mode, execution will terminate before number instructions have been executed if a breakpoint is encountered. number is decimal by default.

- *cmd_list*

  If specified, the list of commands will be executed each time execution stops for any reason. This type of automatic command list is useful for displaying interesting values every time execution stops. If execution stops due to hitting a breakpoint that also has an associated command list, the breakpoint‘s command list is executed first. A *cmd_list* can contain a Go or Step command, in which case program execution will resume automatically.

  It is legal for additional commands to follow a Go or Step command in a command list. They will be stacked for execution in the proper order when execution finally stops for the last time, but this can be confusing. It is recommended that you avoid this situation by ensuring that a Go or Step is the last command to be executed in the list.

  Curly braces surrounding cmd_list are required and the entire Step command must be entered on one line. The current *end execution* cmd_list can be displayed with the BL command. It remains in effect until canceled by a Go or Step command with a new or empty command list (for example, **S {}**).

Examples

```
s 100 {if (@global_var<100) {s 100} {dv "clobbered global_var = %ld\n",
@global_var} }
```

This example executes the program, 100 instructions at a time, until the variable *global_var* is detected to have an invalid value.

Note

This example, like all debugger commands, must be entered on one line.

SP (Stop)

Availability: MON, or a script file under MDI

The SP (Stop) command halts a currently executing program in interactive mode. Interactive mode is normally entered via the go interactive (GI) command and allows a subset of debugger commands to be used while the program is executing.

Usage

SP

Arguments

TC (Target Connect)

Availability: ALL

The TC (Target Connect) command makes a connection to the configured probe and target.

Usage

tc

Arguments

Related Topics

TS (Target Status)

TD (Target Disconnect)

Availability: ALL

The TD (Target Disconnect) command disconnects from a connected target.

Usage

td

Arguments

Related Topics

TS (Target Status)

TS (Target Status)

Availability: ALL

The TS (Target Status) command displays the current target connection status.

Usage

```
ts
```

Arguments

VL (Verify Load)

Availability: ALL

The VL (Verify Load) command is used to verify the program was downloaded correctly. When no arguments are specified, all sections of all files previously downloaded are uploaded and checked against the original executable files.

Usage

VL [[-[N]O *scn_types*] *filename* ...] ...

Arguments

- *scn_types*

  A set of letters specifying section types to verify (**-O**) or not verify (**-NO**):

t	text (program code)
d	data (initialized data)
b	bss (uninitialized data)
l	literals (read-only initialized data)

- *filename*

  If specified, the executable file *filename* is uploaded from the target and verified against the original program file. Otherwise, the previously loaded file(s) is verified. The sections to be verified are determined by *scn_types*.

W (Wait)

Availability: ALL

The W (Wait) command waits for *n* milliseconds before continuing command processing. If *n* is not specified, the default is 100 milleseconds. Wait can be used to introduce delays in the playback of MON script files.

Usage

W

Arguments

! (Execute Operating System Shell)

Availability: MON

The ! (Execute Operating System Shell) command allows you to execute a host operating system command without having to exit the debug monitor.

Note

The **!** must be either last or alone in a cmd_list. In addition, attempts to insert debug monitor comments within the os_command will result in them being sent to, and interpreted by, the operating system with the rest of the os_command text.

Usage

![os_command]

Arguments

- *os_command*

  is any valid operating system command. If it is not supplied, a command shell is started up allowing you to execute any number of host commands, until the OS shell is terminated with an exit command.

MON Operands

This section describes the operands common to many of the commands in the MON command language.

Note
For details on the MON command language see MON Command Basics.

The following is a summary of all operands. For details on the operands, see Operand Details.

- *addr* - **Address**

  {{*number*|**(***expr***)**}[**:***space*]
  sym_name
  expr
  [.]*reg_name*[**.***field*]
  $*ident*

- *cmd_list* - **Command List**

  command [**;** *command*] ...

- *expr* - **Address Expression**

  {*addr*|**(***expr***)**|*expr op expr*|*unary-op expr*}

- *fmt* - **Format**

  {**d**|**u** | **o** | **x** | **X** | **f** | **e** | **E** | **g** | **G** | **c** | **s** | **i**|**I**}

- *ident* - **Identifier**

  {**A..Z**|**a..z**|**_** }[**A..Z**|**a..z**|**0..9**|**$**|**_** |**.**] ...

- *number* - **Constant Number**

  [**0x** | **0o** | **0n**] *digit_string*

- *range* - **Address Range**

  {[*addr*] [**L** *number*]|*addr addr2*|*****[**:***space*]}

- *reg_name* - **Register Name**

  MIPS and ARM Register name tables are provided in reg_name Register Name.

- *space* - **Memory Space**
  MIPS: {**U** | **0** | **1** | **2** | **3** | **R** | **S** | **XU** | **XS** | **XP** | **XK** | **P**|**I**|**D**|**L2** | **DA**}
  ARM: {**P** | **DA**}

  string - **ASCII String**
  "text"

Operand Details

The following pages describe each operand.

addr Address

This operand specifies the location of an object.

Syntax

memory addresses:

```
{number |(expr)}[:space]
sym_name
expr
```

register addresses:

```
[.]reg_name[.field]
```

MON local addresses:

```
$ident
```

Description

This operand specifies the location of an object. An address consists of an offset and a space. An offset is a 32 or 64 bit value giving the byte address of an object relative to the start of an address space (virtual memory, physical memory, general register, and so on).

There are three classifications of addresses: memory addresses, register addresses and “MON local” addresses. For a description of “MON local” address space, see MON Local Variables and Option References. The register addresses reference the processor’s general and special registers, user-defined registers, and optionally specific bit fields within registers.

The memory addresses reference data and instruction memory or memory-mapped devices. These addresses include virtual address segments, and physical (main) memory. The MIPS architecture defines memory in terms of virtual address segments (for example, kseg0, kuseg) mapped into a common physical address space. MON accesses data or instructions either by their physical address **(:P)** or by their virtual address, that can be expressed as an offset from the start of an address segment. (For more information, see the *space* operand in space Address Space Designator.)

Arguments

- *number* - Specifies an offset in bytes from the start of a space. The default base for number is hexadecimal. If an ambiguity arises between a hex digit string, and a sym_name or reg_name, the symbol always takes precedence. In such cases, the hex digit string must be preceded by 0x.
- *expr* - Specifies both offset and a space. An expr must be enclosed in parentheses to allow addition of an explicit space designator. The offset will be aligned to a word boundary for word objects, and to a half word boundary for half word objects. The offset for double words is rounded to either 32 or 64 bits depending on the processor bus width.

- *space* - Specifies the memory address spaces. If a space is not explicitly indicated, a virtual address is assumed. See space Address Space Designator.

  This operand must be appropriate for the command being invoked. For example, the MC command requires physical addresses, so :P is required in that case.

- *sym_name* - An ident specifying the name of a global or static variable, or function in the program being debugged, or a name previously defined via EA (refer to EN (Enter Names)).

- *reg_name* - An ident specifying the name of a processor register. In general, MON recognizes the register names documented in the processor's data sheet, as well as any names read from a Register Definition File. See reg_name Register Name for a list of the register names for your processor.

- *field* - An ident specifying the name of a specific bit field within a register that contains multiple fields. The complete field breakdown is shown when such a register is displayed without the .field qualifier, but the reg_name.field syntax allows a single field to be easily displayed or modified.

- *ident* - An ident naming a MON local variable. The name will be defined and assigned an address the first time it is referenced.

Sometimes an ident might match a valid symbol name and a register name, and might even be a valid hexadecimal number as well. In such cases, it will be interpreted as the symbol name by default, and will need to be prefixed with "." to be interpreted as the register name, or 0x to be interpreted as the hex number. For example, a0 is a sym_name if it exists, while .a0 is always the MIPS reg_name, and 0xa0 is always the number.

An addr that consists of a special register field name (such as SR.IEC) is a special case. It can be used in Display and Enter commands, but it cannot appear in any other context, and ranges of fields are not supported.

Examples

```
1000           // Virtual address 0x1000.
1000:u         // MIPS. Same as above: virtual kuseg address 0x1000.
0n1000         // Virtual address 1000 (decimal).
1000:0         // MIPS. Offset 0x1000 in kseg0 (virtual address
               // 0x80001000 or FFFFFFFF80001000 depending on
               // CPU type).
1000:p         // Physical location 0x1000.
@a0000000      // Location whose virtual address is fetched from a0000000
@0x1234:1      // Location whose virtual address is fetched from
               // offset 1234 in kseg1.
R13            // General Register $13. This register can also be
               // referred to by its software name S0.
lr_irq         // ARM. The IRQ mode link register
CPSR           // ARM.  The current processor status register
SR             // MIPS. The current processor Status Register.
sr.kuc         // The current value of the Kernel/User mode bit.
```

```
pc              // The current Program Counter (unless there is a
                // symbol named “pc”). This is not an actual
                // register, but the address of the next instruction to
                // be executed.
.pc             // Always refers to the Program Counter.
a2              // Register A2 (also can be referred to as .A2, R6,
                // or .R6).
0xa2            // Virtual address 0xa2.
foo_bar         // Location and space defined by symbol foo_bar.
(@.1ptr+5*4)    // ptr is a symbol giving an offset and a space.
                // This expr fetches the byte at that location and
                // adds 20 (decimal) to it.
```

cmd_list Command List

This operand specifies one or more commands to be executed. MON accepts command lists, as well as simple commands, in response to the main prompt or when playing back a command file.

Syntax

command [; *command*]...

Description

This operand specifies one or more commands to be executed. MON accepts command lists, as well as simple commands, in response to the main prompt or when playing back a command file. In this case the commands are executed immediately. Some commands also accept a *cmd_list* enclosed in curly-braces as an operand (such as IF).

Arguments

- *command*

  Is any valid MON command or alias. With the exception of the *fmt* operand, symbolic names, and UNIX filenames; commands and operands are not case sensitive.

The following commands MUST be either the last command or alone in a *cmd_list*: EA, L -c, and !, and any interactive commands (such as "EW" with no list of values).

Command lists (and individual commands themselves within reason) can contain embedded comments as described under "Command Lists" on page 58.

Quiet mode (described under +Q (Enable Quiet Mode), -Q (Disable Quiet Mode)) is automatically turned on while a command alias is being executed. When the alias command is finished, the original state of Quiet mode is restored.

Examples

```
dw pc; dw@PC L 8,i  // show the PC then 8 instructions from the PC

dw cpsr; dw lr_abt; dw @lr_abt-20 @lr_abt,i  // show details surrounding
                                             // data abort exception
```

expr Expression

This operand describes the expressions constructed from addresses.

Syntax

addr

(*expr*)

expr op expr

unary-op expr

Description

This operand describes the expressions constructed from addresses. Expressions combine addresses using the operators listed below. Parenthesized sub-expressions are allowed.

All arithmetic and comparisons are performed in unsigned 64-bit integer mode, even if the operands appear signed. For instance, "`-1`" is treated as the unsigned value "`0xffffffffffffffff`". This also means that the right shift operation always zero fills the high order bits.

The following operators are listed in order of decreasing precedence. Unless modified by parentheses, the associativity of operators of the same precedence is left-to-right except unary operators, which associate right-to-left.

	( )	Parenthesized sub-expressions
unary-op	+ - ~ ! @	Unary plus, unary minus, bit wise complement, logical NOT, address at
	@.{1|2|4|8}	1, 2, 4, or 8 byte value at (indirection)
op	* / %	Multiply, Divide, Modulo
	+ -	Add, Subtract
	<< >>	Left shift, Right shift
	< <= > >=	Relational
	== !=	Equals, Not equals
	&	Bitwise AND
	^	Bitwise XOR
	|	Bitwise OR
	&&	Logical AND
	||	Logical OR

Type information (int, float, pointer-to, and so on) is not available. All numeric operands are assumed to be integers, all arithmetic is performed unsigned, and symbols evaluate to their address. Some symbols are scalar values and evaluate to their constant value. An option enum reference is a good example: $<option_name>.<enum> evaluates to the referenced enum's scalar value.

Note

The indirection operator is “@”, rather than the normal C operator “*”. This is to emphasize that MON does not have the data type information that the C operator requires. “@” means “fetch the address at”, so a full word (or double word, for 64-bit MIPS targets) will always be fetched. @.*digit* fetches a specific datum size of *digit* bytes (1, 2, 4, or 8) from *addr*.

Any reference to a register designator or symbolic name is replaced by the address of the object, not its contents. The first such reference in the expression will also cause the register or symbol’s address space (for example, “General Registers” or “kuseg”) to become the address space associated with the whole expression.

Arguments

Examples

```
main + 20    // Location 32 bytes after the symbol main.
@R2          // Location in memory whose virtual address is in R2.
@PC          // Location of the next instruction to be executed.
             // Indirection through .PC is especially useful. The
             // command “DW @PC L 10,i” will disassemble the 10
             // instructions beginning with the next instruction to be
             // executed.
@RA          // Location in memory whose virtual address is in RA.
@ptr         // Virtual address pointed to by the value at the location
             // defined by the symbol ptr.
(@ptr+5*4)   // ptr is a symbol that describes a location in some
             // address space. This expr fetches the word at that
             // location and adds 20 (decimal) to it.
(@ptr)|8     // Value at ptr or’ed with 8.
```

fmt Data Format

This operand specifies the format used by the Display and Enter commands.

Syntax

{d|u|o|x|X|f|e|E|g|G|c|s|i|I}

Description

The *fmt* operand specifies the format used by the Display and Enter commands.

Arguments

- **d** - Signed decimal integer.
- **u** - Unsigned decimal integer.
- **o** - Unsigned octal integer.
- **x | X** - Unsigned hexadecimal integer. Default is **"X"**.
- **f** - Signed floating point value in decimal notation, with six decimal places.
- **e | E** - Signed floating point value in scientific notation, with six decimal places.
- **g | G** - Signed floating point value in either decimal or scientific notation, whichever is more compact.
- **c** - Single ASCII character.
- **s** - Character string.
- **i**|**I** - Assembled/disassembled instruction (see D (Display Data) and EB, ED, EH, EQ, EW (Enter or Fill Data)).

The case of the x, e, and g formats determines whether alphabetic characters in the formatted data will be in upper or lower case. The fmt operand is the sole exception to the rule that keywords are not case-sensitive in monitor commands.

Some formats are not valid for some object sizes. Refer to Table 4-2 for valid combinations.

Table 4-2. Valid Combinations

Type	**Valid Formats**	
	for Enter	**for Display**
B	**d**, **u**, **o**, **x**, **X**, **c**, **s**	**d**, **u**, **o**, **x**, **X**, **c**, **s**
H	**d**, **u**, **o**, **x**, **X**	**i**, **I**, **d**, **u**, **o**, **x**, **X**
W	any except **c** or **s**	any except **c** or **s**
D	**f**, **e**, **E**, **g**, **G**, **x**, **X**	**f**, **e**, **E**, **g**, **G**, **x**, **X**

See D (Display Data) and EB, ED, EH, EQ, EW (Enter or Fill Data) for examples.

ident Identifier

This operand specifies the name of an entity known to MON.

Syntax

${ A-Z | a-z | $ | _ }[{ A-Z | a-z | 0-9 | $ | _ | . }]...

Description

ident specifies the name of an entity known to MON. The type of entity depends on the context. It can be a symbol, register, command file label, MON local variable, command alias, or trace filter.

Note

ident is case sensitive only for symbol names and MON local and option variable names. For all other uses, *ident* is not case sensitive.

Both options and monitor local variables are referenced with a preceding $. Options have precedence and evaluate to the address of the option. Monitor (debugger) local variables provide temporary storage which does not impact the target memory. Either type may appear anywhere an addr operand is required.

Debugger local variables are created at reference time. A warning is output at creation type if the access is a reference rather than explicit set. Note that at creation time, the referenced size determines the debugger local variable size.

Arguments

Examples

```
MON> dw $foo
Warning: Creating debugger local variable $foo on reference
$foo:  00000000
```

Use the enter command to create and/or assign the value of debugger local var:

```
MON> ew $foo = 0x12345678
MON> dw $foo
$foo:  12345678
```

Note that the address of a debugger local variable is an offset within a reserved memory area of MON. Below is a reference expression to an address (0x260) and its corresponding indirection value. Note that the address is also displayed above, but is converted to the corresponding variable name \"$foo\":

```
MON> dv "%x = %x\n", $foo, @$foo
260 = 12345678
```

Note

'@' is an indirection (dereference) operator. MON does not keep type/size information, so all references must reference the size of the object. The default size is a word for '@' indirection references.

number

This operand is used in address expressions and to specify counts in commands.

Usage

[**0x** | **0n** | **0o**] *digit_string*

Description

The `number` operand is used in address expressions and to specify counts in commands.

Arguments

Examples

```
0x Specifies that digit_string is hexadecimal (base 16), regardless of the
context.
0n Specifies that digit_string is decimal, regardless of the context.
0o Specifies that digit_string is octal (base 8), regardless of the
context.
digit_string — A series of digits in the specified radix, or the default
radix for the context in which number appears.
```

The default number base is hexadecimal for *addr* and *mask*, elsewhere it is decimal. If there is a conflict between a hexadecimal number and a register name or symbolic name, the **0x** base must be explicitly provided.

Note
Unlike standard “C” notation, a leading zero does not specify octal.

```
0o377 == 0ff == 0xff == 0n255 == 255
```

range Address Range

This operand specifies the location of one or more objects in either a memory address space or a register space for the Display/Find, Move, and Enter/Fill commands.

Usage

[*addr*] [**L** *number*]

```
addr addr2
*[:space]
```

Description

The *range* operand specifies the location of one or more objects in either a memory address space or a register space for the Display/Find, Move, and Enter/Fill commands.

```
addr — Specifies the starting address for the range.
number — Decimal number of objects in the range.
addr2 — addr specifying the last address in the range.
The range consists of the objects through and including the object at this
address.
*[:space]— All addresses in the virtual or specified memory space.
```

If *addr* is not supplied, the *range* begins where the *range* of the previous Display or Enter command left off, or at virtual address 0 if this is the first Display or Enter command. If neither "**L** *number*" nor *addr2* is supplied, the *range* consists of a default number of objects.

Arguments

Examples

```
a0000000 a00003ff // 1k starting at virtual address a0000000.
0:P L 1024        // 1k starting at physical address 0.
100:0 L 10        // 10 objects beginning at offset 100 in kseg0 space.
my_ptr            // Default number of objects at offset in space
                  // indicated by the symbol my_ptr.
```

reg_name Register Name

The *reg_name* operand is used to specify the address of any of the processor's internal registers or, in the case of **PC**, the current execution address.

Usage

See the following tables.

Description

The *reg_name* operand is used to specify the address of any of the processor's internal registers or, in the case of **PC**, the current execution address. Some registers have multiple names such as a generic name as well as the specific name defined in the processor architecture manual.

In addition to the predefined names listed below, *reg_name* can be a user-defined register name. If a *reg_name* matches the name of a symbol in the program being debugged, the name must be prefixed with a "**.**" to be recognized as a register name.

Arguments

See Register Definition File for more information.

MIPS Register Names

The MIPS processor contains named registers.

Usage

See Table 4-3 for details of the MIPS register names.

Description

The *reg_name* operand is used to specify the address of any of the processor's internal registers or, in the case of **PC**, the current execution address. Some registers have multiple names such as a generic name as well as the specific name defined in the processor architecture manual.

In addition to the predefined names listed below, *reg_name* can be a user-defined register name. If a *reg_name* matches the name of a symbol in the program being debugged, the name must be prefixed with a "**.**" to be recognized as a register name.

Arguments

See Register Definition File for more information.

Table 4-3. MIPS Register Names

Register Name	Description
r{0..31}	Generic names for the 32 general registers.
zero	Register **r0** (always has the value 0).

Table 4-3. MIPS Register Names (cont.)

Register Name	Description
at	Register **r1** (Assembler Temporary).
v0|v1	Registers **r2** and **r3** (results/expressions).
a0..a3	Registers **r4..r7** (Arguments).
t0..t9	Registers **r8..r15** and **r24..r25** (Temporaries).
s0..s8	Registers **r16..r23** and **r30** (Saved temporaries).
k[t]{0|1}	Registers **r26** and **r27** (Kernel/OS Usage).
gp	Register **r28** (Global data Pointer).
sp	Register **r29** (Stack Pointer).
ra	Register **r30** (Return Address).
mdhi|mdlo	Multiply/divide special registers.
g{0..2}_{0..31}	General registers **$0..$31** for coprocessors 0..2.
c{0..2}_{0..31}	Control registers **$0..$31** for coprocessors 0..2.
fgr{0..31}	Alternate naming convention for Coprocessor 1 general registers **$0..$31**.
fcr{0..31}	Alternate naming convention for Coprocessor 1 control registers **$0..$31**.
{f|d}{0..31}	Alternate naming convention for Coprocessor 1 general registers **$0..$30** (only even numbers are valid). An **f** reference implies single precision, a **d** double precision.
ict{n}, dct{n}, l2t{n}	ICache, DCache and L2 cache tags, where n is the tag entry number (0-N).
sr|cause|epc|prid|index|random|entrylo|context|badvaddr|entryhi|...	System coprocessor (CP0) registers can be referenced by name.
tle#	// TLB entry #Lo, even
tlo#	// TLB entry #Lo, odd
th#	// TLB entry #High
tm#	// TLB entry #Mask
pc	The current Program Counter. This is not an actual register, but the address of the next instruction to be executed.

ARM Cortex-M Register Names

The ARM Cortex-M processor contains named registers.

Usage

See Table 4-4 for details of the ARM Cortex-M register names.

Description

The *reg_name* operand is used to specify the address of any of the processor's internal registers or, in the case of **PC**, the current execution address. Some registers have multiple names such as a generic name as well as the specific name defined in the processor architecture manual.

In addition to the predefined names listed below, *reg_name* can be a user-defined register name. If a *reg_name* matches the name of a symbol in the program being debugged, the name must be prefixed with a "**.**" to be recognized as a register name.

Arguments

See Register Definition File for more information.

Table 4-4. ARM Cortex-M Register Names

Register Name	Description
apsr	Application PSR (APSR) contains the condition code flags.
basepri	Dynamic exception priority control, special-purpose registers. Dynamic management of configurable exceptions supported.
basepri_max	Dynamic exception priority control, special-purpose registers. Dynamic management of configurable exceptions supported.
control	Control Register. Identifies current stack.
eapsr	Execution Application PSR (EAPSR). Accesses EPSR and APSR.
epsr	The Execution PSR (EPSR) contains the Thumb state bit (T-bit).
faultmask	Dynamic exception priority control, special-purpose registers. Dynamic management of configurable exceptions supported.

Table 4-4. ARM Cortex-M Register Names (cont.)

Register Name	Description
iapsr	The Interrupt Application PSR (IAPSR) contains the Interrupt Service Routine (ISR) number of the current exception activation. Accesses IPSR and APSR.
iepsr	The Interrupt Execution PSR (IEPSR) contains the Interrupt Service Routine (ISR) number of the current exception activation. A composite of IPSR and EPSR.
ipsr	The Interrupt PSR (IPSR) contains the Interrupt Service Routine (ISR) number of the current exception activation.
lr	Link Register
msp	Main Stack Pointer. In Handler mode, the processor uses the Main Stack Pointer instead of the Process Stack Pointer.
primask	Register to mask out configurable exceptions. Special-Purpose Priority Mask Register for priority boosting. Manages the prioritization scheme for exceptions and interrupts. When set, raises execution priority to 0.
psp	The Process Stack pointer.
sp	Stack Pointer. The stack pointer used in exception entry and exit is described in the pseudocode sequences of the exception entry and exit.
sp_main	Stack Pointer Main. Active stack pointer
sp_process	Stack Pointer Process. Active stack pointer
xpsr	A composite of all three PSR registers.

ARM Register Names

The ARM processor contains named registers.

Usage

See Table 4-5 for details of the ARM register names.

Description

The *reg_name* operand is used to specify the address of any of the processor's internal registers or, in the case of **PC**, the current execution address. Some registers have multiple names such as a generic name as well as the specific name defined in the processor architecture manual.

In addition to the predefined names listed below, *reg_name* can be a user-defined register name. If a *reg_name* matches the name of a symbol in the program being debugged, the name must be prefixed with a "`.`" to be recognized as a register name.

Arguments

See Register Definition File for more information.

Table 4-5. ARM Register Names

Register Name	Description
cP_R	Coprocessor P register R.
cP_R_oO_M	Coprocessor P register with crn=N, opt2=O and crm=M.
cp15_cntrl, cp15_cpuid, ...	Cp15 registers can also be accessed by name.
r0, r1, r2, r3, r4, r5, r6, r7, cpsr	Unique registers.
r8, r9, r10, r11, r12, r13, sp, r14, lr, r15, pc, spsr	Banked registers, selected by the mode of the processor (as determined by the 5 LSBs currently found in `cpsr`).
r8_fiq, r9_fiq, r10_fiq, r11_fiq, r12_fiq, r13_fiq, r14_fiq, sp_fiq, lr_fiq, spsr_fiq	Selects the `FIQ` mode registers.
r8_user, r9_user, r10_user, r11_user, r12_user, r8_usr, r9_usr, r10_usr, r11_usr, r12_usr	`user` indicates non-`FIQ` mode registers.
r13_user, r14_user, lr_user, sp_user, r13_usr, r14_usr, lr_usr, sp_usr	Selects the `user` mode or `system` mode registers.
r13_svc, r14_svc, sp_svc, lr_svc, spsr_svc	Selects the `supervisor` mode registers.

Table 4-5. ARM Register Names (cont.)

Register Name	Description
r13_irq, r14_irq, sp_irq, lr_irq, spsr_irq	Selects the `IRQ` mode registers.
r13_abort, r14_abort, sp_abort, lr_abort, spsr_abort, r13_abt, r14_abt, sp_abt, lr_abt, spsr_abt	Selects the `Abort` mode registers.
r13_undef, r14_undef, sp_undef, lr_undef, spsr_undef, r13_und, r14_und, lr_und, sp_und, spsr_und	Selects the `Undefined` mode registers.

PowerPC Register Names

The PowerPC processor contains named registers.

Usage

See Table 4-6 for details of the PowerPC register names.

Description

The *reg_name* operand is used to specify the address of any of the processor's internal registers or, in the case of **PC**, the current execution address. Some registers have multiple names such as a generic name as well as the specific name defined in the processor architecture manual.

In addition to the predefined names listed below, *reg_name* can be a user-defined register name. If a *reg_name* matches the name of a symbol in the program being debugged, the name must be prefixed with a "`.`" to be recognized as a register name.

Arguments

See Register Definition File for more information.

Table 4-6. PowerPC Register Names

Register Name	Description
pc	Program counter
r0 - r31	General registers
spr0 - spr1023	Special purpose registers
cr	Condition register

Table 4-6. PowerPC Register Names

Register Name	Description
msr	Machine state register
fpscr	Floating point status and control register
l2mmu_tlb0 - l2mmu_tlb512 l2mmu_cam0 - l2mmu_cam15 (e500v2) l2mmu_cam0 - l2mmu_cam128 (e500mc)	Translation lookaside registers

space Address Space Designator

The *space* operand specifies an explicit address “space” for the address value it is applied to. If no *space* is given, the address value is an offset in the default virtual memory address space. In addition to the default virtual memory space and some special MIPS-specific memory spaces, MON supports the following spaces for all processors.

Table 4-7 and Table 4-8 describe the space designators.

Space Designators For All Processors

There are space designators applicable to all processors.

Table 4-7 details the space designators for all processors.

Table 4-7. Space Designators for All Processors

Space	Processor	Location
:P	ALL	Physical Memory space
:LP	ARM and MIPS Only	Logical to Physical translation space. This is a pseudo space, where the value returned is the physical address corresponding to a given virtual address.
:DA	ARM and MIPS Only	Debug Agent Memory space. This is a pseudo space that accesses special features of the Sourcery Probe.

Space Designators for 32-bit MIPS Processors

There are space designators applicable to the 32-bit MIPS processor.

Table 4-8 details the space designators for 32-bit MIPS processors.

Table 4-8. Space Designators for 32-bit MIPS Processors

Space	Location
:U	offset from kuseg (default): `0`
:0	offset from kseg0: `0x80000000`
:1	offset from kseg1: `0xA0000000`
:2 |:S	offset from kseg2 or ksseg: `0xC0000000`
:3	offset from kseg3: `0xE0000000`
:R	offset from reset vector: `0xBFC00000`
:D	offset in DCache memory
:I	offset in ICache memory

Table 4-8. Space Designators for 32-bit MIPS Processors (cont.)

:L2	offset in L2 cache memory

The MIPS architecture maps several virtual address segments into a common physical address space. These segments are not distinct address spaces in the usual sense. Instead, accessing a memory location through a segment implies: a base address in physical memory, the privilege level required to access the memory, and a cacheable or uncacheable attribute.

Space Designators for 64-bit MIPS Processors

There are space designators applicable to the 64-bit MIPS processor.

Table 4-9 details space designators for 64-bit MIPS processors.

Table 4-9. Space Designators for 64-bit MIPS Processors

Space	Location
:U	offset from kuseg (default): `0`
:XU	offset from xuseg: `0`
:XS	offset from xxseg: `0x4000000000000000`
:XP	offset from xkphys: `0x8000000000000000`
:XK	offset from xkseq: `0xc000000000000000`
:0	offset from kseg0: `0xFFFFFFFF80000000`
:1	offset from kseg1: `0xFFFFFFFFA0000000`
:R	offset from reset vector: `0xFFFFFFFFBFC00000`
:S	offset from sseg: `0xFFFFFFFFC0000000`
:3	offset from sseg: `0xFFFFFFFFE0000000`

Refer to a description of the MIPS RISC architecture for the complete details of memory segments.

The *space* designator tells MON to use the physical space, or to modify the given virtual address by adding the base address of the segment specified. (See addr Address.)

The exact mapping from virtual address to physical address is dependent on the particular processor variant in use.

Space Designators for PowerPC Processors

There is a space designator applicable to the PowerPC processor.

Table 4-10 details the space designator for the PowerPC processor.

Table 4-10. Space Designators for PowerPC Processors

Space	Location
:IN	Read memory using instruction addressing.

string String Literal

Quoted strings are used in string format find (**DB,s**) commands, string format Enter Byte commands, and in the Display Value command.

Usage

"*text*"

Description

Quoted strings are used in string format find (**DB,s**) commands, string format Enter Byte commands, and in the Display Value command.

text is any sequence of printable characters. Non-printable characters can be included by using any of the following C-style “escape sequences”:

\b	backspace (0x08)
\f	formfeed (0x0C)
\n	newline (0x0A)
\r	carriage-return (0x0D)
\t	tab (0x09)
\v	vertical tab (0x0B)
\”	quote
\’	apostrophe
\\\\	backslash
\ooo	octal value (*ooo* is 1 to 3 octal digits)
\x*hh*	hex value (*hh* is 1 or 2 hex digits)

To perform a “**DB,s**” command with no search `value`, or an interactive “**EB,s**” command, MON searches for a null character as the string terminator and automatically inserts a null at the end of a replacement string. But to perform a search for a specific string (**DB,s**= “*string*”) or non-interactive Enter (**EB,s**= “*string*”), MON will neither require a null character for the string to match, nor insert a null automatically at the end of a replacement string. Such strings can be explicitly provided with a null terminator by including **\0** immediately before the closing quote.

Arguments

Assigning Names

Symbolic names representing memory locations or registers can be entered, displayed, or killed in MONICE with the **EN**, **DN**, and **KN** commands. Names are automatically read from the MON information of an executable file when it is downloaded.

MIPS Examples:

```
en prog_base = 1000:1   /* Create a name for the start of program */
en stack_top = FFFFFC:0/* Create a name for the top of stack*/
en prog_ph = 1000:P  /* Create a name for program base
                           physical address */
dn pr*                  /* Display all names starting with 'pr' */
```

ARM Examples:

```
en prog_base = 1000        /* Create a name for the start of program */
en stack_top = FFFFFC
en prog_ph = 1000:P     /* Create a name for program base
                           physical address */
dn pr*                  /* Display all names starting with 'pr' */
```

Note

It is better to assign names for memory-mapped hardware addresses using a Register Definition File. The EN command is meant for symbols within your program.

Command Aliases

The **EA** (Enter Alias) command creates an alias (synonym) for a list of one or more commands. It is normally used to create a short abbreviation for a longer command or sequence of commands that are frequently needed.

Combining **EA** with **MON scripts** allows you to create your own custom commands. A command file can accept parameters, generate output, and supports expression evaluation and flow control for creating loops – everything you need to create your own intelligent command (see **Command Script Files**). The **EA** command allows you to define a name to use for running your command.

Note

Most sample board initialization files, plus the custom init file template, use EA commands to create names for various scripts that are included in the board initialization file.

Aliases can be displayed with the Display Alias (**DA**) command, and removed with the Kill Alias (**KA**) command.

The **DA** (Display Alias) command shows the name and replacement text for one or all currently defined aliases. If the command is entered without a parameter, all aliases are displayed.

The **KA** (Kill Alias) command deletes the name and replacement text for one or all of the currently defined command aliases.

Examples:

```
ea DIA DW @pc L 10,i        // Disassemble instructions from current pc
ea rc fr c                  // Read command file without echo
da DIA                      // Display DIA alias
da *                        // Display all aliases
ka rc                       // Kill rc alias
```

MON Local Variables and Option References

MON local variables provide temporary storage which does not impact the target memory. MON local variables are referenced by a symbolic name that starts with a dollar sign (for example, $temp_var).

Configuration options (see Table 3-9) are also referenced via $option_name. Thus, you cannot have a debugger local variable with the same name as an option name. Note that both short form and long form option names can be used.

These local variable symbols can be used anywhere a normal target address can be used. They are useful in command files for holding expression results or loop counts and the like without intruding on target memory or registers, as in the following sample command file:

```
ew $addr = 0xA0000000/* Create and initialize $addr variable */
ew $write = 0x55555555 /* Create and initialize $write variable */
:LOOP
ew @$addr = @$write /* Dereference $addr and write $write
value there */
mw @$addr, $read /* Dereference $addr and move value to
$read variable */
if (@$read == @$write) { ew $write = ~@$write; goto LOOP }
    // Toggle and loop if value read matched written
dv "ERROR: wrote %X to %X, read %X\n", @$write, @$addr, @$read
   /* display error message */
```

Option variable references can include enum references. The enum reference must include the option name ($<option_name>.<enum> to be valid. Example:

```
if (@$lep == $lep.on) { dv "PC is set at load time\n"}
```

Note

Memory is allocated for MON local variables on an as-needed basis. The first time a MON local symbol name is used, the next available address in the local address space is assigned, the name and address are added to the symbol table, and its size is set. Once referenced, the MON local symbol’s address and size are fixed. Therefore, the first usage should allocate the maximum space needed with an **Enter** command.

Examples:

```
eb $buffer64 L 64 = 0     // Create and clear a 64-byte buffer
```

Formatted Display

The **DV** command (Display Value) allows you to generate formatted output.

The format string controls the operation of **DV** much like the format string in a C printf() statement. Many format controls match those used by printf, but there are differences. Refer to the DV (Display Values) for full details.

Examples:

```
DV "Hello, world!\n"
DV "Byte at %x is %02bx\n", global_char_var, global_char_var
```

Note

Integer formats assume that `int` is 32-bits.

Session Log

An **FW O** (File Write Output) command causes each line printed to the console (including the echo of commands entered) to be logged into the specified file. This allows a permanent record to be made of a debugging session.

Examples:

```
FW O session.log        // Open session.log and enable logging
FW O -                  // Stop logging and close file
FW O +                  // Re-open log file in append mode
```

Note

When using CodeBench, it is recommended to enable logging in the debug launch configuration instead of using the **FW O** command. See Table 2-1.

Command Script Files

Use command script files to extend the MON Command language, automate processing and testing, and so on.

These are commands available for use in scripts. Note that scripts found on the search path are run as commands. For example, the *mmu_dump.maj* script is supplied for most target environments and typing *mmu_dump* at the MON prompt executes it.

Note

Alias and built commands have precedence over scripts.

- FR C - The **FR C** (File Read, Command) command causes MON to read commands from the specified file. When all the commands in the file have been processed, MON resumes reading commands from its previous input source, ultimately returning control back to the console. This is an easy way to input a standard set of commands or to quickly recreate an earlier session. The file can be created manually or logged with the File Write command (**FW C**). If the **FR C** command is part of a multi-command line, any commands following it on the line will be executed after all commands in the new file have been executed. See H (Help).

  Tip: You can call command scripts directly by name, without using the *fr c* command, providing the command script name does not conflict with an existing MON command name.
- H (Help) - This command is used to change the order of command execution when reading commands from a command file. It causes the command file reader to jump to the line following the specified label.
- RETURN - The RETURN command exits processing of the current script or function call.
- SHIFT/UNSHIFT - The SHIFT and UNSHIFT commands change the correspondence between the arguments supplied on an "**FR C** *filename*" command and the formal parameter tokens within the command file.

Command Parameters

Command files can accept and process parameters. Each parameter is an arbitrary string of text delimited by white-space (blank or tab). When each line of the command file is read in, it will be scanned for parameter strings of the form *$$** or *$$n*, where *n* is a one- or two-digit decimal number.

- **$$n** - Parameter replaced with the *n*th parameter of the **FR C** command.
- **$$0** - Special parameter which evaluates to the number of parameters remaining to be processed.
- **$$*** - Special parameter which evaluates to all parameters supplied via the **FR C** command.
- **$$#** - References the pathname of the currently executing *.maj* file.

Note

If no argument was supplied for a particular *$$n* parameter, the parameter will simply be removed from the command line during parameter substitution.

The replacement text can be "pasted" into a larger token by using "\" as a delimiter character.

Examples:

```
FR C example A B C    // Invocation of command file including the following
                      // examples.
$$1\3                 // Replaced with A3
MY\$$2\IDENT          // Replaced with MYBIDENT
temp\$$3              // Replaced with tempC
```

Shift/Unshift Commands

The SHIFT and UNSHIFT commands change the correspondence between the arguments supplied on an “**FR C** *filename*” command and the parameter strings within the command file.

Normally, the first argument is substituted for *$$1*, the second argument for *$$2*, continuing the same pattern. The SHIFT command increments the argument number that corresponds to each parameter number, and decrements *$$0*, effectively shifting the argument array so that a given range of parameter numbers refer to a higher range of arguments. The UNSHIFT command reverses this effect.

Argument shifting is very useful when you want to perform the same series of actions repetitively on an unknown number of arguments (or groups of arguments). The following command file will display the contents of a series of ranges. It expects an argument list of the form: *addr count [addr count]...* On the first iteration, it processes the first two parameters; on the second pass, the next two; and so on.

```
:loop
if ($$0 < 2) { dv "Expected address and count\n";
               goto done }
   dw $$1 L $$2
   shift 2
   if ($$0 >= 2) { goto loop }
:done
```

Related Topics

SHIFT/UNSHIFT

GOTO Command

The **GOTO** command, in combination with debugger local variables and the **IF** command, can be used to construct arbitrary conditional blocks and loops in command files.

The H (Help) command is used to change the order of command execution when reading commands from a command file. It causes the command file reader to jump to the line following the specified label. **GOTO** commands can precede or follow the corresponding label definition. Label definitions and **GOTO** commands have no effect when reading commands from the console, but they will be saved in a command output file if command logging is in effect.

Examples:

```
EW $loop = 0          /* Initialize debugger variable */
:TOP
S 1000 ; DW x         /* Look at variable every so
often*/
IF (@.4x > 0xffff) { GOTO BOOM } /* Has it been trashed
yet? */
EW $loop = @$loop + 1            /* Update loop count */
IF ( @.4$loop < 1000 ){ GOTO TOP}/* Loop, but don't go
forever */
GOTO DONE                     /* Is working, exit cmd
file */
:BOOM                         /* "x" got trashed */
DW $loop                      /* Loop count when we
noticed
                                    trash */
DW x
```

Related Topics

H (Help)

IF Command

The **IF** command supports conditional execution of MON commands. This is extremely useful in breakpoint command lists, where the **Go** command can be used to automatically continue execution if some condition is not met. It is also useful in command files, where the **GOTO** command can be used to conditionally alter the flow of control.

Examples:

```
ew $count = 0
bs foo,10 {dv "entered foo 10 times\n"; if (@.4$count < 0n10) {ew
$count = @.4$count + 1;g@.4$count + 1; g} }
```

The example uses a combination of breakpoint pass counts, the **IF** command, and a MON local variable to implement a breakpoint that will display a message every ten times it is reached, but will not actually stop executing the program until the breakpoint has been reached 100 times.

+/-Q

Enable or disable Quiet mode of command file playback. Normally, MON prompts and commands read from the command file are displayed just as they would be if the commands were entered from the keyboard. But when Quiet mode is active, MON does not display prompts and commands while reading commands from a file.

Quiet mode is automatically turned on while a command alias is used to invoke the command file. When the alias command is finished, the original state of Quiet mode is restored. This allows alias names that invoke command files to look like built-in commands, because there is no extraneous output beyond what the command file explicitly produces.

Chapter 5
MON Command Line Debugger

You can use the MON program as a low-level symbolic assembly debugger. It has a rich set of commands and is uniquely suited to help with hardware bring up and scripted testing.

MON Invocation Line

The MON debugger is started with the mon command, and can use the invocation switches listed below.

```
mon [ [-options] ... [filename] ]
```

Where:

- *options* — is one or more of the options listed in the following table.
- *filename* — is the name of a script file to be run at startup. Normally you would specify your board init script here. MON script files have the extension *.maj*.

At startup MON also searches for the MON script file *startice.maj* and executes it. Although this feature still exists, its usage is being phased out.

Table 5-1. MON Command Line Options

Options	**Description**
-c *cas*	Next parameter is the core access select option (0 - 32). See the Core_Access_Selectoption for more details.
-d *device*	Specifies what probe to connect to. The *device* parameter is the Ethernet hostname or IP address of the probe. When using the virtual probe, this would be the hostname or IP address of the computer running the virtual probe software (See Using the Virtual Sourcery Probe). Note: The space between the switch and *device* is required. On MESP Personal Probes for PowerPC a connection is not based on Ethernet network addresses. Instead, you can use either :0 (if you only have one probe attached, or :<serial #> which can be found on the bottom side of the probe.

Table 5-1. MON Command Line Options (cont.)

Options	Description
-dl	Discovers/displays a list of probes accessible via the network and then quits. Note that discovery can work over subnets if the router is configured to support multi-cast packets.
-dm	Discovers/displays a list of probes accessible via the network and allows you to choose one to connect to and use.
-dn *unit_name*	(Applies to MIPS and ARM Only) Connect to the probe with the given *unit_name*. The default unit name for a Sourcery Probe is the serial number for Sourcery Probe Personal models and FSL*XXXXXX* for Sourcery Probe Professional models where *XXXXXX* is the bottom six digits of the number given on the bottom of the probe. Do not include the colons.
-h	MON switch for displaying help on MON invocation (instead of starting the debugger).
-ni	Non-intrusive startup mode. Normal startup mode resets the processor and clears any breakpoints. Use of **-ni** allows connection to a target without losing this target state information. Also, if the target is currently executing in interactive mode, the debugger will enter interactive mode and not disturb the running program. Note: Because the optional breakpoint command lists are not recorded on the target, they cannot be recovered from the target. However, your original breakpoint will be recovered with an empty command list.
-l	Specifies the target as Little-endian (-l). This switch sets the initial value of the *Trgt_Little_Endian* option to **on**.
-q	Start monitor in Quiet mode (no loading messages, etc.).
-vh	Display the list of CPU types supported by MON.
-v*X*	Specifies the processor type (5KC, ARM922T, ...) that will be debugged. If grouped with other options with a single "-" (dash), the **v***X* option should be the last in that group. For a complete list of processors supported by MON, use the **–vh** option. A specific probe may only support a subset of the listed options. Note: There must be no space between **-v** and the processor type.
-z	Start in *stand-alone* mode. Allows usage of the help system and commands that are not specific to accessing the probe.

Starting MON

There are two reasons to run MON first, even if you intend to use an MDI front end. First, to establish a baseline operation with your Sourcery CodeBench installation and Sourcery Probe configuration files. Second, to create the *`mdi.cfg`* file required when using GDB (outside of CodeBench) with the Sourcery Probe.

To start MON, cd into the directory containing your target initialization script (See Sourcery Probe Target Initialization Scripts), then run MON with suitable **-d**, **-v**, and **-l** switches. For additional information on the MON invocation line, see Table 5-1.

Note

MON is located in the *.../mep/bin* directory of your Sourcery CodeBench installation (See Sourcery Probe Software Support). If this directory is not in your PATH environment variable, then you will need to prepend the path before the mon program name.

Examples:

```
MON -vh // Show the supported CPU version (-v) switches
MON -v5Kc -d 205.158.243.236 // Ethernet ( d ___), big endian (no -l)
MON -vA8 -dm -l // Cortex-A8 (-vA8), little endian (-l), list discoverable
probes (-dm)
```

Note

A space is required between the -d switch and the port name or IP address, but a space is prohibited in the -v switch. The little endian (-l) switch is a lower case L, not a number one.

When MON starts, it displays a number of messages with version and configuration information, then it displays a MON> prompt. Assuming there were no errors in your startup command files, at this point you should be able to access and view both registers and memory on your target. You can even download, step through, and run a program. MON Command Language provides numerous examples of using the MON command language and details the syntax of each command and parameter type. A hierarchical help system is available with the H command.

Creating a MON Connection

This section provides you with the steps required to create a MON connection.

Prerequisites

- The probe has been configured with an IP address. See Mentor Embedded Sourcery Probe Introduction.
- A connection to the evaluation board is being established with an existing *.maj* board initialization file.

Procedure

1. Change directories to the directory containing the mon (monice) executable as shown in the following example.

Note
If you are using Windows® 64-bit, you must use "Program Files(x86)" in the path.

```
C:\>cd <install_directory>i686-mingw32\arm-none-eabi\mep\bin
```

Tip: See Table 1-4 for default installation directory locations, then look in the *tsp/_templates* subdirectory.

2. Scan for the available probes on the network and create a connection:

```
>mon -dm -vARM920T -l
..\tsp\arm\atmel_at91rm9200ek\at91rm9200ek.maj
```

Where in this example:

-dm - discovers probes available on the network

-vARM920T - specifies the processor type

-l - specifies the little endian processor configuration

..\tsp\arm\atmel_at91rm9200ek\at91rm9200ek.maj - specifies the relative path to an installed board configuration file

The output for the probe discovery will be similar to the following:

```
Symbolic Assembly Level Debug Monitor, version V7.0.12 - Win32
Copyright (c) 1987-2011 by Mentor Graphics Corporation - All Rights
Reserved.
Reading command history from: 'C:\Documents and
Settings\kwilliam\Application Da
ta\mon_mon.hist'
Scanning for visible Probes...
    Model:                 Serial #: IP Addr:        UnitName:
0: MESP-Personal / ARM07380127     169.254.156.159 jpsmeuj
1: MESP-Pro / ARMFSL00FF73         134.86.101.72   FSL00FF73
Select a Probe 0..1, q>
```

3. Enter the number of the probe you want to connect to at the **Select a Probe** prompt, or enter **q** to quit without connecting to any probe.

NOTE: To connect to a probe without using probe discovery, use the -d<ip_addr> command:

```
 >mon -d 134.86.101.72 -vARM920T -l
..\tsp\arm\atmel_at91rm9200ek\at91rm9200ek.maj
```

where in this example:

-d 134.86.101.72 - specifies the probe’s IP address

Results

Output similar to the following is produced:

```
Symbolic Assembly Level Debug Monitor, version V7.0.12 - Win32
Copyright (c) 1987-2011 by Mentor Graphics Corporation - All Rights
Reserved.
Processing register file: C:\Program
Files\CodeSourcery\Sourcery_CodeBench_for_A
RM_EABI\i686-mingw32\arm-none-eabi\mep\bin\arm/spaces.rd
Processing register file: C:\Program
Files\CodeSourcery\Sourcery_CodeBench_for_A
RM_EABI\i686-mingw32\arm-none-eabi\mep\bin\arm//arm920t.rd
Processing register file: C:\Program
Files\CodeSourcery\Sourcery_CodeBench_for_A
RM_EABI\i686-mingw32\arm-none-eabi\mep\bin\arm/\armbase32.rd
Reading command history from: 'C:\Documents and
Settings\kwilliam\Application Da
ta\mon_mon.hist'
Establishing communications with probe: 134.86.101.72...
Select returned ready
Connection verified
Processing register file: C:\Program
Files\CodeSourcery\Sourcery_CodeBench_for_A
RM_EABI\i686-mingw32\arm-none-eabi\mep\bin\arm\mds.rd
Target System:  MESP-Pro / COP, S/N FSL00FF73
Firmware Rev:   2.1.0 build 1
Hardware Rev:   1:7:8:0
Unit Name:      FSL00FF73
Ethernet MAC:   00:04:9F:00:FF:73
Ethernet IP:    134.86.101.72, Subnet Mask: 255.255.254.0, (Static)
Target CPU:     ARM920T
Connected via:  TCP/IP, Device name: 134.86.101.72
Target Endian:  little

Reading commands from C:\Program
Files\CodeSourcery\Sourcery_CodeBench_for_ARM_E
ABI\i686-mingw32\arm-none-
eabi\mep\bin\..\tsp\arm\atmel_at91rm9200ek\at91rm9200e
k.maj
MON> +q  // Enter quiet mode
Reading at91rm9200ek.maj ...
ice_jtag_clock_freq = 0.009
Target power detected
Auto JTAG detection process detected 1 TAP
JTAG bypass test passed
DCache=16k, ICache=16k
Target is Halted
Executing RTI script function
ice_jtag_clock_freq = 0.009
Auto JTAG detection process detected 1 TAP
Initializing target...
ice_jtag_clock_freq = 18.75
Target initialization completed.
Finished reading at91rm9200ek.maj.
```

To confirm the connection, use the d r0 command as shown in the following example:

```
MON> d r0
.r0         2000138C
```

Command Line Editor

This section describes the keys used to perform command line editing in MONICE, and the MON console window provided by most of the EDT debugger interface libraries. These special keys provide a convenient way to edit command lines and recall recently entered command lines.

- <Ins>—(Insert) Toggles between Insert and Over-type modes. In Insert mode, normal characters are inserted at the current cursor position. In Over-type mode, normal characters replace the character at the current cursor position. On MS-DOS systems the cursor size reflects the mode. Insert mode is a half field block, Over-type mode is an underline. UNIX systems do not support cursor size changes.
- <BS> — (Backspace) Deletes the character to the left of the current cursor position.
- <Del> — (Delete) Deletes the character at the current cursor position.
- <Up> — (Up Arrow) Replaces the current line (if any) with the previous line in the circular buffer.
- <Down> — (Down Arrow) Replaces the current line (if any) with the next line in the circular buffer.
- <Left> — (Left Arrow) Moves the cursor to the left one character.
- <Right> — (Right Arrow) Moves the cursor to the right one character.
- <Home> — Moves the cursor to the beginning of the current line.
- <End> — Moves the cursor to the end of the current line.
- <PgUp>—(Page Up) Replaces the current line (if any) with the first (oldest) line in the circular buffer.
- <PgDn> — (Page Down) Replaces the current line (if any) with the last (most recent) line in the circular buffer.
- <C-PgUp> — (Control-Page Up) Deletes the entire contents of the circular buffer.
- <C-PgDn> — (Control-Page Down) Deletes the currently selected line (if any) from the circular buffer.
- <Esc> — (Escape) Deletes all text from the current line. The circular buffer is not affected.
- <Enter> — (Return) Enters the current line as input to MON. The cursor does not have to be at the end of the line.

- <F1> — (Function key F1) Entered once, searches the circular buffer for a line whose beginning matches the text typed so far. The search starts from the last (most recent) entry in the buffer. If a match is found, the matching line replaces the current line. If a match is found, <F1> can be hit again to find the next match for the original text.

Note

Many Unix consoles have different shell modes which can alter or filter out the keyboard codes used for command line editing and history recall. If these keys seem not to perform their function, try switching to a different mode. Specifically on Sun machines be sure to use a "shell tool" rather than a "cmd tool." "Cmd tool" does not work properly, even with scrolling disabled.

MON Help Command

A summary of MON commands is displayed in response to the help (H) command with no parameters, and help on a specific topic is available with the H <topic> command.

See MON Help for details. Also, a summary of these commands is listed in MON Command Quick Reference.

MON is a powerful command line and scripting language for accessing and exercising the processor and target system.

Chapter 6
Meta Debug Interface (MDI)

This section describes the Mentor Graphics® Meta Debug Interface (MDI) library for the Sourcery Probe line of debug probes. It is directed to end-users who intend to use the probes with any debugger that supports the MDI specification.

Note

Except where explicitly stated to the contrary, the term Sourcery Probe refers to all models in the Mentor Embedded Sourcery Probe series.

If you want to implement an MDI-compliant debugger and/or an MDI library, refer to the MDI specification document in the */mep/mdi* directory of your Sourcery CodeBench installation.

Tip: See Table 1-4 for default installation directory locations, then look in the *tsp/_templates* subdirectory.

What is MDI?

The Meta Debug Interface (MDI) is an Application Programming Interface (API) that defines a standard set of data structures and functions that abstract hardware for debugging purposes. Having a standard meta interface allows debuggers and debug agent tools (ROM resident debug monitors, ICEs, JTAG probes, and so on) from different vendors to work together.

The initial MDI specification, */mep/mdi/mdispec.pdf*, (See Table 1-4 for default installation directory locations) was jointly developed by Embedded Performance, Inc. (subsequently acquired by Mentor Graphics Corporation) and LSI Logic Corporation. Mentor Graphics makes the MDI specification freely available, and welcomes its adoption by any interested vendor. While the initial implementations targeted MIPS processors, the specification is architecture-neutral, so you can adapt it to other architectures.

The MDI API is implemented as a shared library (Windows DLL file or Linux *.so* file), called an MDILib. The MDILib implements the standard MDI API on top of the vendor-specific mechanism for communicating with the actual debug agent which controls the target system. As such, the MDILib is normally provided by the vendor of the debug agent tool that provides target access and control services (typically an ICE, debug monitor, or simulator).

Similarly, the debugger typically supports MDI via a layer of code that implements the debugger's proprietary API or communications protocol by making calls on the functions exported by the MDILib. This translation layer may be included within the debugger executable or it may be in a separate module.

MDI Versions

Over time, there might be updated versions of the MDI specification document. If there is any functional change to the API, the MDI version will be updated. The MDI API includes a mechanism for the debugger and library to negotiate the actual MDI API version that will be used. It is intended that a debugger or library should be able to support older versions of the API as well as the version that was current when it was implemented.

Any debugger or library claiming MDI compliance should list the range of MDI API versions with which it is compatible. If the supported version range of the debugger and library overlap at all, they should work correctly together. The released version of the MDI API is 1.3, and the Sourcery Probe MDI library supports this version.

Note

The specification document itself also has a version number (currently 1.3), but it is the API version that matters when checking for interoperability.

Getting Started with MDI

This section explains the steps to configure the Sourcery Probe MDI interface library to work with an MDI complaint debugger.

Note

This section is intended for users wishing to access the probe via a programmable API and is not directly applicable to the general CodeBench users.

Installation and Setup

Follow the procedure below before configuration of the MDI environment.

Procedure

1. Install Sourcery CodeBench and your MDI compliant debugger. This manual assumes that you have already installed both packages.
2. Use MON to create MDI configuration files. See Advanced Configuration.
3. Consult your MDI client tool’s documentation and follow its setup instructions to configure access to the MDI library.

For information on setting up the Sourcery Probe hardware and additional details on the configuration process and advanced configuration options, see Mentor Embedded Sourcery Probe Introduction.

Configuring the MDI Environment

Sourcery CodeBench includes the MDI interface library software that provides the connection from your debugger to Mentor Embedded Sourcery Probe. It also includes sample startup files for standard reference boards, and on-line documentation. So the first step is to install the Sourcery CodeBench on your computer.

Introduction

To configure MDI, run MON and use that to create the MDI configuration files.

Note
See Advanced Configuration for more details.

MDI Files

An MDI Configuration consists of the following files:

- *mdi.dll* (Windows) or *mdi.so* (Linux)

  The MDI interface library that will be loaded by your third party debugger to provide the connection to your Sourcery Probe. This file will be copied to the MDI configuration/project directory. Alternatively, your debugger may allow you to specify the full path name of the MDI library file, in which case it can be run from the Sourcery CodeBench installation directory (recommended). See your debugger's documentation for more details.

- *mdi.cfg*

  MON creates an *mdi.cfg* file specific to your configuration (Sourcery Probe and startup file), and saves it in the MDI configuration directory. This file contains configuration information required by the MDI library.

A sample *mdi.cfg* file showing how you can combine multiple MDI configurations into a single configuration file is provided in the `/mdi` folder of your Sourcery CodeBench installation. See Sourcery Probe Default Installation Directories for the installation directory for your specific architecture and OS.

- `./mep/mdi/mdispec.pdf`

  The MDI specification which describes the API in great detail. It is provided for informational purposes. You do not need to read this document in order to use an MDI compliant debugger and library.

- `./mep/mdi/mdi.h`, `./mep/mdi/mdimips.h`, `./mep/mdi/mdiarm.h`, `./mep/mdi/mdiload.c`

  C source code, headers for the MDI API and sample debugger code to load an MDI library. These files are of interest only to those actually implementing an MDI library or an MDI compliant debugger.

- *<target>.maj*

  The Sourcery Probe and target hardware configuration information is provided by the *<target>.maj* target initialization script. This is typically named after the board for which it is intended. Sample startup command scripts for standard reference platforms are included in the `/tsp/<arch>/` directory of the Sourcery CodeBench software installation. See Sourcery Probe Default Installation Directories for the installation directory for your specific architecture and OS. If no suitable *<target>.maj* script is available, you can select the sample template (*template.maj*) and edit it by hand. See Sourcery Probe Target Initialization Scripts.

- In addition, many of the files found in the tree containing the MDI shared library are referenced and used by MDI.

The Debug Environment

The following sections outline the basic steps necessary to configure the debug environment using MON.

Because every target system is different, the Sourcery Probe needs information about the target system design to operate correctly. The primary file, called *<target>.maj*, configures the JTAG interface, certain capabilities of the Sourcery Probe, minimally initializes the target hardware, and declares memory for the board.

Procedure

1. Create a new, empty directory for staging the configuration files, then cd into that directory. For example:

```
mkdir SourceryProbe
cd SourceryProbe
```

2. Sourcery CodeBench includes pre-validated startup files for many common reference platforms in the */tsp* directory of your Sourcery CodeBench installation. See Sourcery Probe Default Installation Directories for the installation directory for your specific architecture and OS. If you are using a standard reference platform, you can simply copy the corresponding files into the *SourceryProbe* configuration directory you created in step 1. If your board is similar to a standard reference platform, you may need to adjust it to account for the hardware differences by editing it with your preferred text editor. See Adapting a Reference Board Initialization Script to Your Board.

```
cp <install_dir>/mep/tsp/<arch>/<target>/*.maj . /* using arm or mips as
appropriate */
```

If your board is your own design, you will need to create suitable startup files. Copy the template file from the *<install_dir>/<host>/<target>/mep/tsp/<arch>/_template* directory, then fill in the details, as explained in the comment blocks and *readme.txt* file.

Creating the MDI Configuration File

The MDI shared library is responsible for managing the details of the Sourcery Probe configuration. It uses the same *target.maj* script as MON. However, CPU information and communication parameters are set via the *mdi.cfg* configuration file instead of using command line switches. The easiest way to create *mdi.cfg* is to start MON as described above and then ask MON to create the file for you.

Procedure

1. CD into the working directory created in The Debug Environment.
2. Start the MON command line debugger as described in MON Invocation Line.
3. Use the following command to create an *mdi.cfg* file in your current working directory that matches the configuration in use by MON.

```
MON> fw mdi mdi.cfg
```

Note

The *mdi.cfg* file created in this way supports just one specific Sourcery Probe and target. If you have more than one Sourcery Probe and/or target system, you can edit the *mdi.cfg* file to define multiple devices and controllers, and then pick the desired configuration for each debug session.

Advanced Configuration

This section describes advanced topics related to MDI configuration.

MDI Configuration File (mdi.cfg)

The rest of this section is for users wishing to create a more advanced configuration *mdi.cfg* file. Some reasons why you might want to do this: To enable connection to multiple Sourcery Probe's and/or targets via one configuration file. In such a case a debugger is required to query you as to which Sourcery Probe configuration to connect to at startup time.

The MDI Launch Process

When the debugger first connects to the MDI library, the library will try to open the *`mdi.cfg`* file by looking first in the current working directory, then in the directory containing the library (Windows version only), then in the directories specified in your PATH environment variable. It is usually most convenient to have a single *mdi.cfg* file located in the same directory as the library.

The MDI specification requires the debugger to query the MDI library for a list of devices that are available, and to give you a way to select which device to connect to. In the case of the MDI for Sourcery Probe, there is also the possibility that there is more than one physical Sourcery Probe available to which to connect. The MDI configuration file (*mdi.cfg*) provides the necessary device and probe identification information to the MDILib, so that it can respond to the debugger's query request.

MDI Configuration File Format

The MDI configuration file is an ASCII file containing comments (C++ style) and keyword-value pairs. Keywords are not case-sensitive. There are two types of values, strings and numbers. Strings must be enclosed in double quotes ("), unless they are valid identifiers (start with an alphabetic or underscore, and contain only alphanumerics and underscores). Numbers and quoted strings conform to C/C++ language syntax.

The sample configuration file uses line breaks and indentation to improve readability, but these are not required. As long as there is at least one whitespace character (space, tab, or line break) between all keywords and values the file will be processed correctly.

Organization

The configuration file is organized into sections.

Each section defines Global settings, a Device, a Controller, or the MDIDeviceList. Conventionally, the Global section is first, followed by all the Device sections, followed by Controller sections and finally the MDIDeviceList section. But the only requirement is that the MDIDeviceList section follows all the Device and Controller sections.

Global

The Global section defines values for configuration settings that are not associated with a particular target device or connection. This section is optional, but it is strongly recommended to include it and specify the PATH value. The syntax is as follows:

```
Define Global
       PATH        PATHString
       LogFile     LogPathString
       CommandFile CmdPathString
       ConnectMode TargetConnectMode
```

- *PATHString* is the full path name of the directory into which Sourcery CodeBench was installed. It is used to locate various files needed by the MDILib without having to add specific directories to the PATH environment variable.
- *LogPathString* is the full or relative path name of a file which will have detailed MDI session log information and console output written to it. This entry should only be used when requested by Mentor Graphics technical support.
- *CmdPathString* is the full or relative path name of a default target initialization script (normally <*target*>*.maj*) that will be processed automatically when a device is opened which does not specify a more specific CommandFile value in its MDIDeviceList, Device, or Controller definitions.
- The *CommandFile* entry is optional in every definition section, but at least one applicable value is required for every DevNameString defined in the MDIDeviceList section. If multiple values are available, the order of precedence is the value specified in the DevNameString definition itself, followed by the value specified in the referenced Device definition, followed by the referenced Controller definition, followed by the Global definition.

  CommandFile is normally specified in the Device definition, since there are commonly different configuration commands needed for different devices. Specifying a default in the Global definition is useful if you have several devices sharing a common startup command file.
- *TargetConnectMode* is one of the following: Default, Reset, Halt, Run. Reset tells MDI to reset the target at target connection time. Halt interrupts the target at connect time, but does not reset it. Run connects to the target, but does not disturb its state. Default is the same as Reset.

Device

Each Device section defines a target device and includes the CPU type, memory organization, and the various identification strings required by the MDI Specification. If you have more than one target type with different CPU types or memory organizations, you must provide multiple Device sections to define them. The syntax is as follows:

```
Define Device        DevIDString
```

```
Family         FamilyString
Class          ClassString
ISA            ISAString
Part           PartString
Vendor         VendorString
VendorFamily   VendorFamilyString
VendorPart     VendorPartString
VendorPartRev  VendorPartRevisionString
VendorPartData VendorPartDataString
Endian         { Big | Little }
Cpuid          CPUString
CommandFile    CmdPathString
CoreAccessSel  { 0...32 }
```

- *DevIDString* is an arbitrary string value used to identify the Device section when it is referenced from the MDIDeviceList section.
- *CPUString* is a string value giving the specific CPU type identifier. This is the same value specified by the -v command line option when running MON (see MON Command Language).
- The *Endian* value specifies the target system's memory organization (big-endian or little-endian).
- *CmdPathString* is the full or relative path name of a startup command file (normally *target.maj*) that will be processed automatically when a device is opened which does not specify a more specific CommandFile value in its MDIDeviceList definition. If CommandFile is specified here, the value overrides values specified in the Global and Controller definitions.
- *CoreAccessSel* corresponds to MON's -c option. This optional parameter is valid only on multi-core environments and signals the core to which to connect. Note that this option can be overridden in the startup script referenced in the command file. A value of 0 is the same as no selection. This setting is also valid in the MDIDeviceList section. A value here overrides any setting provided in a specific MDIDeviceList device.

Note

Any setting in Device overrides a setting here.

All of the other values are string values that are passed to the debugger to identify the CPU type. The intent of the MDI specification is that each chip vendor will document standard values for these strings for their parts. That may or may not happen, but the Sourcery CodeBench MDI library makes no use of these values. You can set these values to whatever strings your debugger is expecting, if any. Otherwise, the values are arbitrary.

Note

Only the Endian and Cpuid values are required to be present in a Device section. The Family and Class values are set to the MDI-specified strings by default, so they can always be omitted. The rest are set to the string NotSet by default, and can be omitted unless your debugger expects them to be set to a particular value (which it must document).

Controller

Each Controller section defines a particular Sourcery Probe to connect to and includes the communication port (serial port or Ethernet host name), baud rate, and optionally the startup command file to load for initialization. If you have more than one Sourcery Probe, or more than one startup initialization file, you must provide multiple Controller sections to define them. The syntax is as follows:

```
Define Controller ControllerIDString
         Port        PortString
         CommandFile CmdPathString
         ConnectMode TargetConnectMode
```

- *ControllerIDString* is an arbitrary string value used to identify the Controller section when it is referenced from the MDIDeviceList section.
- *PortString* is a string value that provides the Ethernet hostname or Ethernet IP address to use to connect to the Sourcery Probe. When using the virtual probe, this would be the hostname or IP address of the computer running the virtual probe software (See Using the Virtual Sourcery Probe).This is the same value specified by the -d command line option when running MON (see MON Command Language). On MESP Personal Probes for PowerPC a connection is not based on Ethernet network addresses. Instead, you can use either :0 (if you only have one probe attached, or :<serial #> which can be found on the bottom side of the probe.
- *CmdPathString* is the full or relative path name of a target initialization script (normally *<target>.maj*) that will be processed automatically when a device is opened which does not specify a more specific CommandFile value in its MDIDeviceList or Device definition. If CommandFile is specified here, the value overrides a value specified in the Global definition.
- *TargetConnectMode* is one of the following: Default, Reset, Halt, Run. Reset tells MDI to reset the target at target connection time. Halt interrupts the target at connect time, but does not reset it. Run connects to the target, but does not disturb its state. Default is the same as Reset.

  If a ConnectMode reference is specified here, the value overrides a value specified in the Global definition.

MDIDeviceList

The last section in the configuration file is the MDIDeviceList section. It defines the MDI device name strings that the debugger will display, and associates each name with a specific Device and Controller section. The syntax is as follows:

```
Define MDIDeviceList
      { DevNameString
         Device      DevIDString
         Controller  ControllerIDString
         CommandFile CmdPathString
         CoreAccessSel  { 0...32 }
}....
```

If you have more than one Device or Controller section, the MDIDeviceList section will contain multiple sets of DevNameString + Device + Controller entries to list and name all the valid combinations of Device and Controller.

- *DevNameString* is a string value to be passed to the debugger to identify a particular MDI Device that the debugger can connect to. If there is only one *DevNameString* entry in the MDIDeviceList section, the debugger may just automatically open the device. Otherwise, the debugger should present you with a list of the DevNameString values and let you select value to which to connect.
- *CmdPathString* is the full or relative path name of a startup command file (normally *target.maj*) that will be processed automatically when the device is opened. If CommandFile is specified here, the value overrides values specified in the Global, Controller, and Device definitions.
- *CoreAccessSel* corresponds to MON's -c option. This optional parameter is valid only on multi-core environments and signals the core to which to connect. Note that this option can be overridden in the startup script referenced in *CmdPathString* above. A value of 0 is the same as no selection. Also, note that any setting in the Device section overrides a setting here.

Startup Command File

The Sourcery Probe has a number of configuration options that must be set correctly for the target system. These options are set by debugger commands, and Sourcery CodeBench debuggers automatically look for a target initialization script named *<target>.maj* at start up. The Sourcery CodeBench includes a library of sample startup files for standard reference boards.

Note
See Sourcery Probe Target Initialization Scripts

When using a third-party debugger with the Sourcery Probe MDI library, the name of the startup command file is specified in the MDI configuration file. A full path name can be provided, or a relative path name, or just the file name. If the full path is not specified, the MDI library will try to open the file by looking first relative to the current working directory, then the directory containing the library (Windows version only), then the *./mep/bin* subdirectory of the Sourcery CodeBench installation directory if PATH is specified, then the directories specified in your PATH environment variable. The directory containing the startup command file will also be added to the list of directories to search for other files.

Note

The MDI configuration file created by MON always uses <*target*>*.maj* as the target initialization script name. But, if one configuration file needs to describe dissimilar targets, then separate target initialization scripts are required, and your custom initialization script will need to reference the different file or path names.

Troubleshooting

There are two troubleshooting techniques you can use if you are having problems using an MDI-compliant third party debugger with the Sourcery Probe.

The first is to try to reproduce the problem using the MON debugger. This eliminates the third party debugger and its possibly non-compliant use of the MDI API from the equation. Make sure that MON is loading the same *target.maj* file as the MDILib so the initial configuration will be the same. If you have similar symptoms when using MON, then it must be a target board or Sourcery Probe issue (most likely a configuration problem), rather than an MDI issue.

MDI Interface Logging

If your Sourcery Probe appears to be operating properly when using MON, the next question is whether the culprit is the MDILib or the debugger itself. This may not be easy to determine from the visible symptoms, so the MDILib includes a logging facility that can record all of the MDI operations requested by the debugger and their results. The log file will provide the detailed information needed by customer support to see exactly what is going wrong with the debug session.

Logging can be enabled via two different methods (a change to your MDI configuration file, or an environment variable). Both methods are described in the following sections.

Setup Logging via the Configuration File

This is accomplished by editing the *mdi.cfg* file that is in use by your debugger of choice. In this file, you'll find a section named Define Global as shown in the following example. There should already be some options specified under this global section.

```
Define Global
      PATH ...
      CommandFile ...
```

To enable logging, add the LogFile option as shown in the following example.

```
LogFile LogPathString
```

- *LogPathString* is the full or relative path name of a file which will have detailed MDI session log information and console output written to it. If spaces are used in the path or filename, then the path needs to be quoted. The path can contain either "\" or "/" as directory separators, but backslashes need to be doubled (C expression rules). See the following examples:

```
c:\\temp\\my_log.txt
```

or

```
/usr/my_log.txt
```

The details presented here are also covered in the section describing the configuration file, but these instructions act as a focal point if you simply need to enable logging.

Setup Logging via an Environment Variable

Another way to enable MDI session logging is to define an environment variable named MEPLOG whose value is the pathname of the file which will contain the log output. It is also possible to set `MEPLOG` to `"console"`, in which case the log output is sent to the debugger via its MDICBOutput() callback function. Generally, logging to a file is the preferable method.

Interoperability

The MDI specification includes both required and optional services. Any debugger or library implementation claiming MDI compliance must support all required services and document the optional services it supports. This chapter provides this information and other details about the Sourcery CodeBench MDILib implementation for Sourcery Probe that may affect interoperability with MDI-compliant debuggers.

Required Services

The Sourcery CodeBench MDILib implements all required MDILib services. However, there are a few cases where the MDI Specification leaves aspects of the implementation of a required service up to the MDILib, or where the implementation does not exactly match the specification. These cases are detailed in the following sections.

Most services defined by the MDI specification are provided by functions in the MDILib that are called by the debugger. There are also a few services that the debugger is required to provide for use by the MDILib, via callback functions. We list the ones that the Sourcery CodeBench MDILib actually uses.

MDICacheFlush

This function allows the debugger to request that the processor's Instruction and/or Data caches be flushed and/or invalidated. MDILib supports this service by passing the request on to the Sourcery Probe. But what the Sourcery Probe does with it can vary depending on the capabilities of the actual CPU. For example, it may not be possible to flush the cache without also invalidating it.

MDIRunState

This function returns the current status of the target system (running, hit breakpoint, and so on) to the debugger. The MDI Specification requires the debugger to call MDIRunState() frequently when the target is running and not running.

The Sourcery Probe may occasionally send status or event notification messages when the processor is not running, such as when target power is turned on or off. So the MDILib does depend on the debugger calling MDIRunState() when the target is not running, to process these events in a timely manner. If the debugger does not make these calls, notifications may be delayed until the next MDI service is requested or can even be lost entirely.

MDIRead

This function is called by the debugger to read the contents of memory and registers. The debugger passes an address, an object size, and a count of the number of objects to read. An address consists of an offset and a "resource". Resources are values that identify specific memory and register address types, and are architecture-specific. MDILib support for some resources is optional. To allow the debugger to determine whether a particular resource is supported, the MDI Specification assigns special meaning to MDIRead() calls with a count of zero. When the requested count is zero, the MDILib is required to just check the address and return a success or error status, based on whether the address is valid and supported.

Currently, the Sourcery Probe does not provide a way to directly query whether an address is valid for the current target. Instead, it generates an error message when a transfer to an invalid or unsupported address is actually attempted. So the MDILib uses built-in knowledge of the CPU type to determine whether to return a success status for MDIRead() calls with a count of zero. This works well enough for typical usage, like whether or not there are floating point registers or cache tag registers. But there are cases, like cores that allow optional coprocessors to be added, where the MDILib can not know whether the target has the corresponding registers or not. In such cases, it will return success to the zero count query, but actual reads and writes may fail after displaying an error message.

MDIDoCommand

This function is called by the debugger to cause the MDILib to execute an ASCII command using its internal command interpreter. Recognizing that no debugger API can possibly abstract all possible features of all possible debug tools, the MDI Specification allows the MDILib to provide additional functionality via a command interpreter. MDIDoCommand() is unusual in that it is an optional service, but if it is provided by the MDILib then the debugger is required to use it. That is, the debugger must provide a way for you to enter arbitrary command lines that the debugger will pass to MDIDoCommand().

The MDILib does implement an extensive command language, and, therefore, relies on the debugger to support MDIDoCommand(). Nearly all the MON commands described in the MON Command Language are available in the MDILib as well, with the notable exceptions being breakpoint and execution commands.

MDICBInput, MDICBOutput

These are required callback functions provided by the debugger. MDICBOutput() allows the MDILib to pass strings that the debugger must display, while MDICBInput() allows the MDILib to get keyboard input from you.

The MDILib relies on MDICBOutput() to display informational and warning messages, target program output generated via the EPI-OS or semi-hosting features supported by MIPS and ARM Sourcery Probe, and output generated by commands passed to MDIDoCommand(). It also relies on MDICBInput() to get input for the target program, interactive commands passed to MDIDoCommand(), and sometimes responses to error or warning message prompts. If the debugger does not provide these required services, it is not MDI-compliant and it will not be able to connect to the MDILib successfully.

Optional MDI Services

The MDI Specification defines a number of optional services that an MDILib may support, and some required services have optional aspects. MDILibs are required to document what they actually implement for all optional behavior. These cases are detailed in the following sections for the MDI for Sourcery Probe.

MDIOpen

This function is called by the debugger to establish a connection to a particular target device. It must be called before any debug services (read, write, execute, and so on) can be performed. The debugger passes a parameter indicating whether it wants exclusive or shared access to the device. To enable various types of multi-processor debugging, the MDI specification permits one or more debuggers to open multiple devices at the same time, and to even open the same device multiple times. However, since the capabilities of debug tools varies widely, an MDILib is not required to support multiple simultaneous connections.

Target Groups

The MDI specification includes an abstraction for the concept of having multiple target devices treated as a group. If connections are made to multiple target devices in the same group, execution can be started and stopped on all of them with a single service request. Support for the Target Group services is optional.

Since the MDILib does not support connecting to more than one target device at a time, it also does not support Target Group services.

Target access while running

The MDI specification allows the debugger to make calls on MDI service functions even while the target is executing code. But for most services, it also allows the MDILib to return an error status if it does not support the particular service while the target is running.

Note that it may be necessary for the Sourcery Probe to pause execution while performing a given debug service request, but will avoid doing so whenever possible. For example, some processors support a method of memory access that does not require halting the processor.

MDIFind

This function is called by the debugger to search for a value or pattern. The MDI Specification requires that the MDILib support this service for patterns up to 256 objects long, but only for memory (as opposed to registers). Since an object can be up to eight bytes in size, a pattern can be as long as 2048 bytes with an optional mask of equal size.

The MDILib supports MDIFind() only for memory address ranges, not registers. Also, Sourcery Probe limits the total length of the search pattern to about 1400 bytes (700 bytes when masked). If this limit is exceeded, MDIFind() returns a `"not found"` status. For object sizes of four and eight bytes, this is a technical violation of the MDI specification since we support less than the required pattern length of 256 objects. But this is very unlikely to be a problem in practice. Most debuggers only support searching for a single value, so our worst-case search pattern limit of 88 masked double words should be more than adequate.

MDICacheQuery

This function may be called by the debugger to retrieve the attributes of the caches present on the target device, if any. In the MDI Specification support for this function is optional.

The MDILib uses the Sourcery Probe cache configuration options trgt_cache_type, trgt_icache_*, and trgt_dcache_* to provide the requested information. Whether these options are available depends on the specific processor type being used. If they are not available, MDICacheQuery() does not return any information.

MDIReset

This function is called by the debugger to reset the target device. The MDI Specification defines four types of reset operation that can be requested: MDIFullReset (reset entire target system, processor and board), MDIDeviceReset (reset processor, including peripherals in SoC devices), MDICPUReset (reset processor only), and MDIPeripheralReset (reset peripherals only in SoC devices). Support for multiple types of reset is optional for both the debugger and the MDILib.

The Sourcery Probe provides flexible support for reset operations. It supports two types of reset requests, "reset target" and "reset processor". The MDILib maps the MDI reset types as follows: If the debugger requests MDIFullReset, the MDILib performs a target reset. MDIDeviceReset and MDICPUReset both cause the MDILib to perform a processor reset. Since the Sourcery Probe does not provide a mechanism for resetting peripheral logic without also resetting the CPU, the MDILib does nothing if an MDIPeripheralReset is requested.

MDISetBp

This function is called by the debugger to set hardware and software breakpoints and triggers. The MDI specification provides a fairly ambitious abstraction for breakpoints, including permanent and temporary software (standard) breakpoints, and hardware breakpoints on instruction execution, data load and/or store access, or bus transaction. Hardware breakpoints can include an address range, data value, and a mask for the data value. They can also be specified to generate a trigger signal as well as or instead of halting the processor.

Obviously, the actual ability of debug systems to support all these types of breakpoints can vary widely. An MDILib is only required to support the software breakpoint types. All hardware breakpoint support is optional.

All of the breakpoint options are supported by the MDILib in the sense that they are passed on to the Sourcery Probe. The Sourcery Probe, in turn, supports all of the hardware breakpoint and external trigger capabilities of the particular processor.

Trace services

The MDI specification provides a set of optional services that can be used by the debugger to enable the capturing of execution history trace data (instructions executed and possibly data accesses), and to fetch the captured data for display. When used with a probe supporting trace, the MDILib supports all MDI Trace services.

MDICBPeriodic

This is an optional callback function that may be provided by the debugger. If provided, the MDI Specification requires that the MDILib call this function at least once every 100 milliseconds while processing a long-running MDI function to give the debugger an opportunity to process user interface events. The specification does not specify a limit on how frequently MDICBPeriodic may be called.

If it is provided, the MDILib actually calls the MDICBPeriodic function much more frequently than the required 100 milliseconds, and will call it at least once even during services that are completed in less than 100 milliseconds.

MDICBLookup, MDICBEvaluate

These are optional callback functions that may be provided by the debugger. MDICBLookup() allows the MDILib to ask the debugger what symbol or source line corresponds to an address, if any. MDICBEvalute() allows the MDILib to ask the debugger to evaluate an expression using the debugger's rules, and return a value or address. These callbacks are intended to be useful to MDILibs that implement a command language, since they may not have access to symbolic information for the program being debugged.

Currently, the MDILib does not call MDICBEvaluate(). It will use MDICBLookup() if it is available.

Command Interpreter

The MDILib for Sourcery Probe implements the MON command language interpreter in order to provide full access to all the capabilities of the Sourcery Probe, not just the subset that is supported by the MDI API and the debugger.

As described in Required Services, the debugger is required by the MDI specification to provide a way for you to enter commands and data to, and see the output from, the MDILib's command interpreter. When using Sourcery CodeBench, an MDI window is created and monitor commands (which can be abbreviated as mdi) can be used to execute MDI commands.

The MDILib command interpreter supports nearly all of the MON command language, as documented in MON Command Language. All data display and enter commands, including trace data display, reset commands, and configuration commands are available. The command file and alias commands that allow the command language to be extended are also available.

Note

If the `MDILib` can not find the `./mep/bin` installation directory, then the MON Help command will not work and some register definitions may be missing. Specifying the `PATH` value in the Global section of the `mdi.cfg` file is recommended for this reason. For Windows and cygwin environments, the directory will be found if the `mdi.dll` file is being loaded from there. Otherwise, the `./mep/bin` directory must be added to your `PATH`.

Chapter 7
Using the Virtual Sourcery Probe

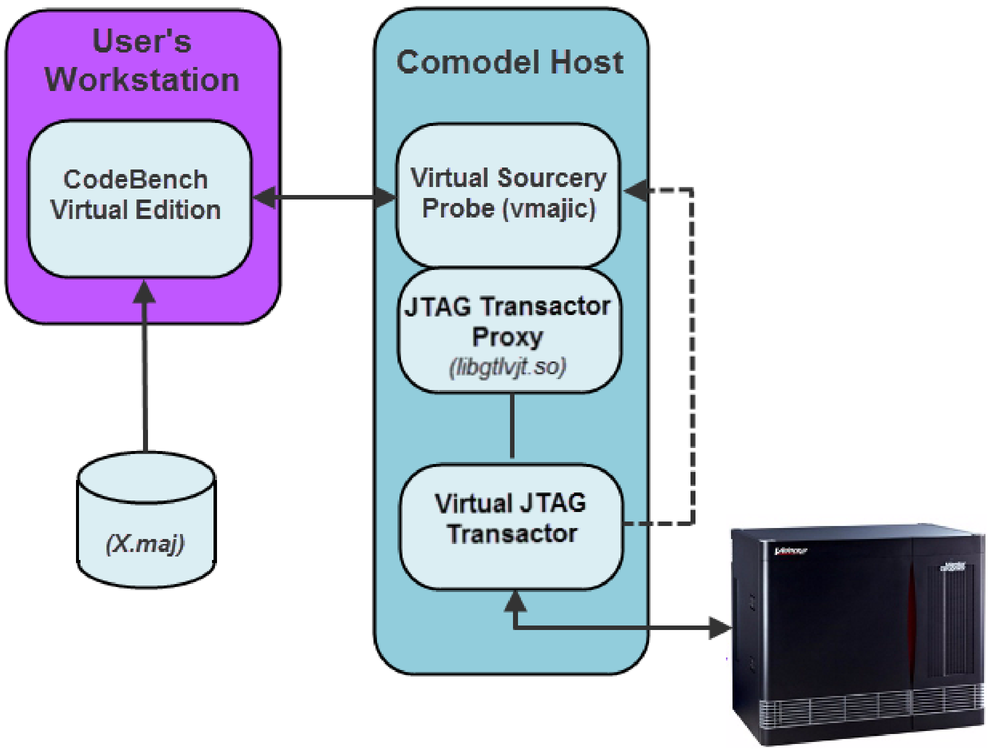
The traditional Sourcery Probe debug environment is comprised of debugger software running on a PC, connected to a hardware probe via an Ethernet or USB connection. The Sourcery Probe connects to the system under test via JTAG (plus some dedicated special function I/O signals). The debugger software provides the user interface, and implements high level debug features using debug services provided by the Sourcery Probe.

With the Virtual Sourcery Probe (also known as VMAJIC), the debug server software runs on a PC instead of dedicated probe hardware, providing an interface to the virtual JTAG port of a processor model running on a hardware emulator. This enables software debugging to begin on the actual hardware design long before any SOC devices are produced, allowing hardware and software to be brought up together. The advantages of hardware/software co-debug include:

- Ability to test software on an actual hardware design before the hardware is available.
- Ability to exercise hardware using realistic software scenarios instead of manually generating test vectors.
- The emulation software provides visibility into the hardware operation and the interactions between the hardware and software that’s not possible with actual hardware.
- Ability to debug your software in a familiar development environment in which you can perform tasks such as run control and single stepping and view variables, registers, and memory in the source code.
- Ability to view test bench waveforms and use assertions to catch defects in the hardware.
- Schedule reduction by overlapping hardware and software development phases.
- Quality improvement by fixing hardware defects before fabrication instead of working around hardware defects after fabrication.

The Virtual Sourcery Probe uses a JTAG Transactor API to access the JTAG interface of a hardware simulation/emulation model of the Design Under Test (DUT). The JTAG Transactor option for the Veloce® Emulator interfaces to the virtual JTAG port using TestBench Xpress (TBX). It has a HDL part written in System Verilog, which is integrated into the hardware design, and a SW part written in C++ to which the virtual probe connects.

Figure 7-1. The Virtual Sourcery Probe Development Environment

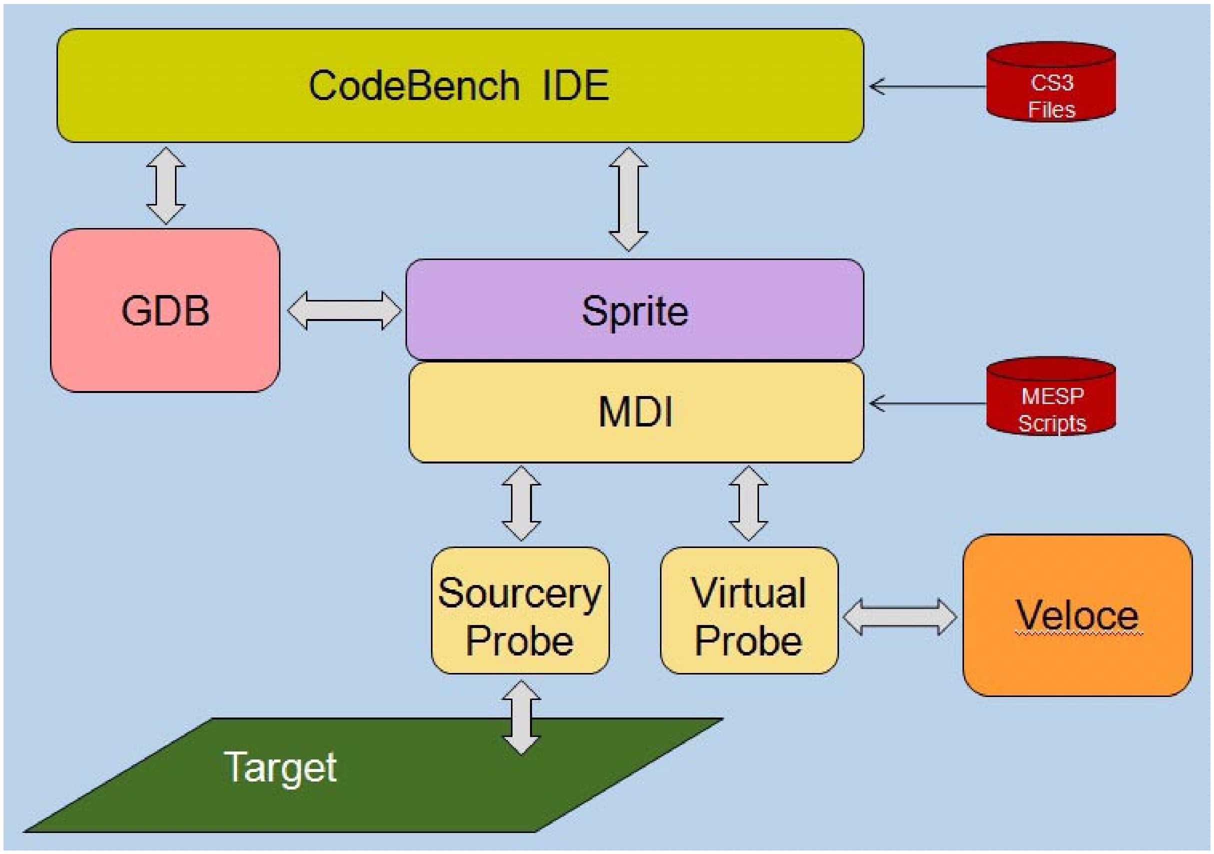


Sourcery Debug Environment

Several components comprise the Sourcery debug environment:

- Sourcery CodeBench is an Eclipse/CDT-based development environment, using GDB as the core debugger.
- The Sprite is a CodeBench component that provides a GDB server debug interface, using MDI to communicate to the Virtual Sourcery Probes.
- The MAJIC Debug Server software within the Sourcery Probes (whether real or virtual) controls the processor and queries its state via a JTAG connection to the processor's debug controller.

Figure 7-2. Sourcery Debug Block Diagram



Veloce Debugging

Refer to the *Veloce Virtual Probe User Guide* for information on the JTAG Transactor and TestBench Xpress™.

In order to start a MESP/Virtual software debug session you need access to the following software and hardware:

- The Veloce Software Debug Suite must be installed on the comodel host. This includes the virtual probe and JTAG transactor.
- CodeBench Virtual Edition installed on the user's work station.
- An executable to be run or debugged on the device under test.
- A custom initialization script appropriate for the design. See “Creating a Custom Initialization Script Using a Template” on page 21.

Starting a Veloce Emulation Run

Use TestBench Xpress to download the hardware design into the emulator and begin the emulation run. This is analogous to powering on the target system when debugging on actual hardware.

Procedure

1. Log on to the comodel host machine of the Veloce emulator and go to the TBX compiled database directory. Set the environment for TBX and Veloce software as explained in the "Starting a Debug Session" section of the *Veloce Virtual Probe User Guide*.

Note

The hardware emulation may be run from a script file, or from the VeloceGUI software. Please see the "Starting a Debug Session" section of the *Veloce Virtual Probe User Guide* for more information on loading and running the emulation model.

2. Wait for the following message to appear:

```
"Veloce Virtual JTAG Server started. Listening at Port Number <port>"
```

3. Configure a CodeBench debug launch for Sourcery Probe, as described in the *CodeBench Getting Started Guide*, and use that to launch the debugger.

Results

At this point, the design has been downloaded on to the emulator and the MESP/Virtual JTAG Transactor Server component is waiting for the client to initiate a connection.

MESP/Virtual Invocation Line

If multiple users each want to use a vJTAG within their design on the same Veloce comodel host concurrently, then each user requires unique *vmajic* and Virtual JTAG Transactor (VJT) instances. Therefore, the transactor port number is not always the same, and a different port number is assigned on each run. Normally the transactor automatically starts *vmajic*, but if you launch *vmajic* manually then you must provide the transactor port number using the MESP/Virtual program -d command line switch (detailed below). The port number is then displayed in the emulation start-up transcript.

Note

Only use the MESP/Virtual invocation line in special cases when directed to do so by Mentor Graphics technical support. MESP/Virtual is normally started automatically by the Veloce testbench (see the *Veloce Virtual Probe User Guide*).

If a design has multiple instances of vJTAG then only one transactor instance is required, but a separate *vmajic* instance is required for each instance of vJTAG. In this case the vJTAG instance name to debug must be provided. This is achieved by assigning each vJTAG instance and associated socket connection with a unique port number and socket instance pathname, using the MESP/Virtual program -d command line switch.

The MESP/Virtual program provides several command line switches to control its behavior, as shown in Example 7-1.

Note
To quit MESP/Virtual, just press ^**C**, or use the Sourcery Probe setup menu.

Example 7-1. The MESP/Virtual Program Command-line Switches

```
./vmajic [-d [host][:port][+instpath]] [-h] [-j lib] [-s] [-c] [-z]
```

Each MESP/Virtual program command line switch is detailed in Table 7-1.

Table 7-1. MESP/Virtual Command Line Switches

Command Line Switch	Description
-d [host] [:port] [+instpath]	***host*** — the device string specific to the virtual transactor library. For the Veloce JTAG transactor, this is the IP address or hostname of the PC running the TBX software in Figure 7-1. Without the -d switch, MESP/Virtual assumes the JTAG transactor is on the local host. Therefore, the host name can be omitted if MESP/Virtual is running on the Veloce comodel host. ***:port*** — a required parameter providing the Virtual JTAG Transactor (VJT) port number. ***+instpath*** — specifies the instance path within the design that is associated with this socket connection. This field is optional if the design has only one vJTAG instance, but required if the design has multiple vJTAG instances. It is not permitted for multiple probes to connect to the same vJTAG instance.
-h	Displays the Help text for each MESP/Virtual command-line switch.
-j lib	***lib*** — the file name (and optional path) of the virtual JTAG Transactor library. For example, *-j $VSDS_HOME/vprobe*
-s	Automatically display the MESP/Virtual setup menu on the console. If omitted, the setup menu can be brought up by pressing the <ESC> key in the MESP/Virtual console.

Table 7-1. MESP/Virtual Command Line Switches (cont.)

Command Line Switch	Description
-c	Disables the *vmajic* console. This switch is used when the *vmajic* debug server is automatically launched.
-t	Using this switch causes the Veloce emulation to terminate when MESP/Virtual shuts down. Without the -t switch, MESP/Virtual disconnects from the Veloce and leaves the emulation running when it shuts down. Note: The terminate/disconnect mode can now be selected in the debugger with the **Ice_Shutdown_On_Exit** option. See Table 3-9 on page 42 for details.
-z	Start MESP/Virtual in stand-alone mode – it does not load any JTAG transactor library, and you can do very little with it but verify uplink communication to the debugger.

Example 7-2. MESP/Virtual Manual Invocation

Invoke MESP/Virtual from the Veloce Software Debug Suite *bin* directory:

```
cd $VSDS_HOME/vprobe/bin
```

Tip: See Table 7-1 for default installation directory locations, then look in the *tsp/_templates* subdirectory.

Use the following command:

```
./vmajic -j $VSDS_HOME/vprobe/lib/linux/libJTagClient.so \
-d host:port+inst
```

Figure 7-3. MESP/Virtual Boot Screen

```
Mentor Embedded Sourcery Probe

    ____Configuration Data____
    Probe Model         : MESP-Virtual
    Serial Number       : 8323329
    Unit Name           : VMAJIC
    Software Version    : 2.3.0 build 33, Jun  5 2012
    Ethernet Name       : hostname
    Ethernet IP         : xxx.xxx.xxx.xxx
    Ethernet MAC        : XX:XX:XX:XX:XX:XX
    Ethernet subnet     : 255.255.254.0
    JTAG Transactor     : velcsdcomodel

Loading Virtual JTAG transactor library: libJTagClient.so
vjt_id.vmajic_id_str:   MESP-Virtual v2.3.0 build 33, Jun  5 2012
```

```
        vjt_id.vjt_id_str: Veloce VJTAG Transactor
```

At this point a connection has been created between the MESP/Virtual and the JTAG Transactor component. Now MESP/Virtual will wait for a debug connection request.

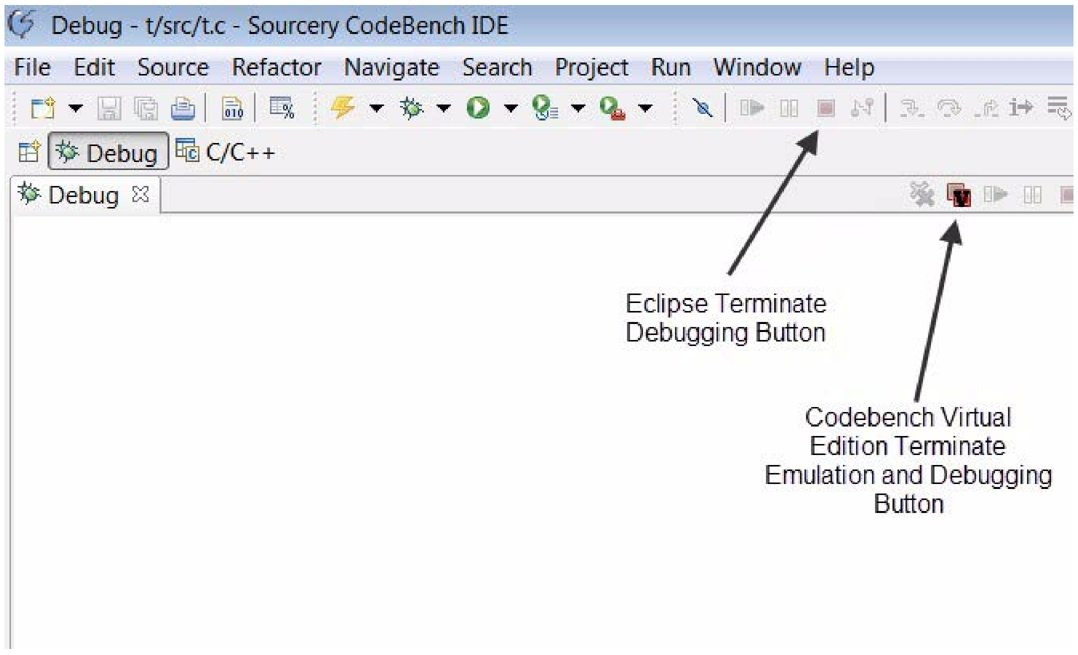
Normally there is no need to interact with the MESP/Virtual program directly, as the user interface is implemented by the debugger software. You may use the Sourcery Probe Setup menu in the MESP/Virtual terminal window to terminate the MESP/Virtual program. The emulation model may be terminated or left disconnected based on the Sourcery Probe Setup menu option.

Ending a Debug Session

You can end a debug session as follows:

- The standard Eclipse Terminate button (see Figure 7-4) will terminate the debug session in CodeBench, but will leave the emulation job and the VMAJIC debug server running. You can then relaunch the debugger for a new session.
- The CodeBench Virtual Edition also provides a special Terminate Emulation and Debugging button (see Figure 7-4) that shuts down the emulation job and the VMAJIC debug server, as well as the CodeBench debugger. This frees up the Emulator when it's no longer needed.

Figure 7-4. Codebench Virtual Edition Terminate Emulation and Debug Button


Debug - t/src/t.c - Sourcery CodeBench IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
C/C++
Debug
Eclipse Terminate Debugging Button
Codebench Virtual Edition Terminate Emulation and Debugging Button

Note

Setting the **Ice_Shutdown_On_Exit** configuration option to yes allows you to automatically shut down VMAJIC and the Veloce emulation job when finished debugging. It does not terminate immediately, but after all debug sessions are subsequently terminated, then the emulation job and VMAJIC debug server will be terminated as well. See Table 3-9 on page 42 for details.

Related Topics

MESP/Virtual Invocation Line

Hardware/Software Cross Triggering

One of the advantages of HW/SW co-debug with the Virtual Sourcery Probe is the ability to trigger a software break based on an event in the hardware (which may or may not even be visible to the software), or to trigger the hardware debugger based on an event in the software. The following sections provide details on these features.

Overview of Cross Triggering from Virtual Probe

Cross triggering Veloce from the Virtual Probe provides a means to automatically capture hardware waveforms on Veloce when software execution stops at a breakpoint. This makes it easy for a software engineer to capture hardware activity at the point of suspected failure which the hardware engineer can then review.

Triggering in this way is accomplished by setting a Veloce trigger on a signal within the CPU core that indicates the CPU has entered debug mode. With such a trigger set, software execution can proceed normally, and as soon as a breakpoint is hit, or execution is manually stopped, the Veloce waveform is triggered to capture what was happening in the hardware at that moment in time.

See the *Veloce Virtual Probe User Guide* for specific steps and examples.

Overview of Cross Triggering from Veloce

Cross triggering the Virtual Probe from Veloce provides a means to automatically stop software execution when an event of interest occurs in the hardware, even when the hardware event is not something directly visible to the software or CPU. This makes it easy to see where the code is, and inspect the software state, when the given hardware event occurs.

Triggering in this way is accomplished by setting a Veloce trigger on the hardware event of interest. When the event occurs, the trigger matures, and emulation halts so the hardware state can be inspected from the Veloce environment. At this point the software environment is frozen because the entire emulation, including the CPU, is paused.

To look at the software side, resume emulation with a debug request signal raised at the CPU. This stops software execution just like a breakpoint would, and the software debugger is then able to inspect the state of the software.

It is also possible to automatically resume emulation with the CPU in debug mode if the goal is to simply stop the software and not inspect hardware waveforms or state information

See the *Veloce Virtual Probe User Guide* for specific steps and examples.

Appendix A
Probe Settings and Updates

The following sections describe how to set up your Sourcery Probe device using Sourcery CodeBench.

Updating the Sourcery Probe Firmware Using CodeBench

This section describes the steps required to update the firmware that is running on Sourcery Probe Professional and Sourcery Probe Personal.

Procedure

1. Start Sourcery CodeBench per the instructions in the Sourcery CodeBench Getting Started manual. Close the Welcome screen if it appears.

2. Select **Run** > **Sourcery Probe** > **Settings and Firmware Update**.
   The Settings and Firmware Update Dialog Box appears.

3. Enter the IP address or hostname in the **Probe IP/Hostname** field for the probe, and click **Retrieve Settings**. If you do not know the IP address or hostname of your probe, click **Discover** to view a list of all Sourcery Probe devices currently available on the local subnet and select your probe. See Sourcery Probe Discovery Dialog Box.

4. Select the Firmware tab from the Settings and Firmware Update dialog box. See Firmware Tab (See Figure A-4).

5. Select a valid Update file.

Table A-1. Valid Update Files for Sourcery Probes

Probe Name	Valid Update File
Sourcery Probe Professional -- OR -- Mentor Embedded GIGA-JTAG Probe	mesp-professional_vXYZbN.mud

Table A-1. Valid Update Files for Sourcery Probes

Probe Name	Valid Update File
Sourcery Probe Personal -- OR -- Mentor Embedded USB-JTAG Probe	mesp-personal_vXYZbN.mud

6. Click **Program Update**. This updates the firmware on the Sourcery Probe.

Caution
Do not unplug or otherwise disturb the probe during the update process. Wait for the process to complete before moving on.

7. Click **Close** to close the dialog box.

Related Topics

Updating the Sourcery Probe Firmware from the Command Line

Sourcery Probe Personal Settings (ARM and MIPS Only)

Sourcery Probe Professional Settings

Settings and Firmware Update Dialog Box

Sourcery Probe Discovery Dialog Box

Updating the Sourcery Probe Firmware from the Command Line

Run the `mep_update` console application to update the firmware for the Sourcery Probes from the command line.

The *mep_update* utility is in the */mep/bin* directory within the Sourcery CodeBench installation. See Table 1-4 for default installation directory locations.

For a default Sourcery CodeBench for ARM EABI installation on Windows, this is at *<install_dir>/i686-mingw32/arm-none-eabi/mep/bin.*

Prerequisites

- You must be using an ARM or MIPS model of Mentor Embedded Sourcery Probe Professional or Personal.

Procedure

1. Change to the */mep/bin* directory of your Sourcery CodeBench installation.

2. Issue the `mep_update` command. The `mep_update` command discovers probes visible on your network. It enables you to select a probe, and searches for probe firmware updates within your installation. Example output from `mep_update`:

```
MEP Update Ver 0.1.0 Copyright 2010 Mentor Graphics Inc.

Scanning for visible Probes...
    Model:              Serial #:   IP Addr:          UnitName:
 0: MESP-Pro / ARM      FSL01E3A8   134.86.101.113    FSL01E3A8
 1: MESP-Pro / ARM      FSL01E3B2   134.86.101.197    FSL01E3B2
                           Hostname: alm101dyn197.alm.mentorg.com
 2: Giga JTAG / COP     FLS00FF7A   134.86.100.185    FSL00FF7A
                           Hostname: alm100dyn185.alm.mentorg.com
 3: MESP-Pro / COP      FSL00FF97   134.86.100.135    FSL00FF97
                           Hostname: alm100dyn135.alm.mentorg.com
 4: Giga JTAG / COP     FLS00FF90   134.86.100.186    FSL00FF90
                           Hostname: alm100dyn186.alm.mentorg.com
 5: MESP-Pro / ARM      FSL00FF73   134.86.101.72     FSL00FF73

Select a Probe 0..5, q>
```

Tip: To exit the mep_update console, enter **q** at the command prompt.

3. Select a probe from the list of probes that `mep_update` displays. For example, to select the MESP-Personal /ARM probe, enter **0** at the command prompt. This displays the current firmware version for the selected probe and displays a list of available firmware update files (*.mud*).

4. Select the appropriate firmware from the displayed list and press **Enter** to begin the update. For a list of valid update files for your probe, see "Valid Update Files for Sourcery Probes" on page 191.

   If the update is successful, a message similar to the following displays:

```
..........
Downloaded 563200 bytes in 0.000 sec.
Programming image - Checksum 0x0210f5a3
Firmware image update completed successfully
Probe Update is complete.
Please cycle the power of your probe to activate the new firmware
```

Examples

The following example demonstrates how to update the firmware for a probe on a Linux host machine.

```
$./mep_update
MEP Update Ver 0.1.0 Copyright 2010 Mentor Graphics Inc.

Scanning for visible Probes...
    Model:           Serial #:     IP Addr:         UnitName:
 0: Giga JTAG / COP  FSL00FF65     134.86.178.63    FSL00FF65
                                   Hostname: mep3.sje.mentorg.com
```

```
Select a Probe 0..0, q> 0

Probe update (.mud) files found:
    0: ../mep_firmware/mesp-personal_v225b3.mud
    1: ../mep_firmware/mesp-professional_v225b3.mud

Choose a probe update file from the list (0..1, q): 0
  Processing Update file ../mep_firmware/mesp-personal_v225b3.mud header:
  # majic update file
  type = fw_app p6.1
  checksum = 0x99de3c4a
  # revision: 2.2.5 build 3 date: fri jan 13 11:49:20 2012


  Downloading majic update from file: ../mep_firmware/mesp-
personal_v225b3.mud
  .............
  Downloaded 842732 bytes in 0.682 sec.
  Programming image - Checksum 0x99de3c4a

  Firmware image update completed successfully

  Probe Update is complete.
```

Related Topics

Updating the Sourcery Probe Firmware Using CodeBench

Sourcery Probe Personal Settings (ARM and MIPS Only)

Sourcery Probe Personal models for ARM and MIPS use TCP/IP sockets over USB for communicating with the debugger. This requires compatible IP addresses on the probe and PC. By default, the USB probe automatically uses an IP address in the Local Link range 169.254.x.x.

Note

Sourcery Probe Personal models for PowerPC do not require any communication settings because a dedicated USB driver is used.

In most cases it is best to leave the probe set to use Local Link mode so that it can pick an IP address for itself.

- If you are using Windows 7, then local link mode is fully supported, and no special configuration is required.
- If you are using Windows XP or Linux, then you will need to assign a static IP address to the PC network adapter. Refer to “Connecting to the Target and Host Computer” in the *Mentor Embedded Sourcery Probe Personal Hardware Manual* for this procedure.

Although no special probe settings are normally required, in some cases you might want to set the probe to a static IP address in a different address range. There are two ways to do this:

- Use the Sourcery Probe Console to configure the settings. This is the best method if you cannot establish a network connection to the probe. See “Sourcery Probe Console” in the *Mentor Embedded Sourcery Probe Personal Hardware Manual* for details on this procedure.
- Use the Sourcery Probe Settings feature of CodeBench (assuming you can establish a network connection). See Changing the Settings Using CodeBench.

Changing the Settings Using CodeBench

You can change the Sourcery Probe settings using CodeBench by following the Procedure below.

Prerequisites

- You must be using an ARM or MIPS model of Sourcery Probe Personal.

Caution

New static IP settings can result in the probe becoming inaccessible if the settings are incorrect or incompatible with the network adapter on your PC. If the probe is inaccessible after programming new settings, you must change your network adapter settings to match the IP range used by the probe.

Procedure

1. Start Sourcery CodeBench per the instructions in the Sourcery CodeBench user manual. Close the Welcome screen if it appears.
2. Select **Run** > **Sourcery Probe** > **Settings and Firmware Update**.
   The Settings and Firmware Update Dialog Box appears.
3. Enter the IP address or hostname in the **Probe IP/Hostname** field for the probe, and click **Retrieve Settings**.

   Tip: If you do not know the IP address or hostname of your probe, click **Discover** to view a list of all Sourcery Probe devices currently available on the local subnet.

   This opens the Sourcery Probe Discovery Dialog Box and scans all available network adapters for Sourcery Probes. Note that not all Sourcery Probe devices can be discovered by this dialog box.
4. The Settings and Firmware Update dialog box displays the current settings for the selected probe device.

5. Choose one of the following options on the Settings tab:
   NOTE: If you need to use multiple Sourcery Probe Personal probes on the same host, please contact Mentor Graphics for help with your setup.

   a. Select **Use DHCP/Local Link** for the probe to obtain the IP address configuration automatically from the DHCP server on the network. This is the default configuration.

   b. Select **Static IP** to use the static IP address configuration.

      This requires you to enter the IP address and subnet mask. Since this is a private network to your PC, you can choose the IP address setup that makes sense for your environment. Be careful not to collide with existing network adapter settings on your host computer. We suggest using something in the range of 192.168.X.X with a subnet mask of 255.255.255.0.

6. After you make the necessary changes to the settings, click **Program Settings** to set them on the device and reboot it.

Related Topics

Settings and Firmware Update Dialog Box

Sourcery Probe Discovery Dialog Box

Updating the Sourcery Probe Firmware Using CodeBench

Configuring a Sourcery Probe Network Interface for Local Link Addressing

The Linux service "Network Manager" is responsible for automatically configuring host network interfaces. The steps in this section describe how to recognize and configure a Sourcery Probe network interface for Local Link addressing.

Note

The screenshots and instructions provided here are from the Gnome Network Manager version available with Ubuntu 10.04. Network manager is evolving rapidly and the UI is known to look different in earlier and later versions, but the configuration settings are the same – configure the interface for Local Link IPv4. A similar Network Manager GUI is available for the KDE UI. There is a new command line interface for Network Manager "nmcli", but it does not yet have the ability to configure interfaces as needed here.

Procedure

1. Make sure avahi-autoipd is installed. Most modern Linux distributions have this installed by default.

```
Installation on Debian/Debian derived Distros
```

```
# sudo apt-get install avahi-autoipd
Reading package lists... Done
Building dependency tree
Reading state information... Done
avahi-autoipd is already the newest version
```

2. Configure the host Ethernet interface for Link Link Only:

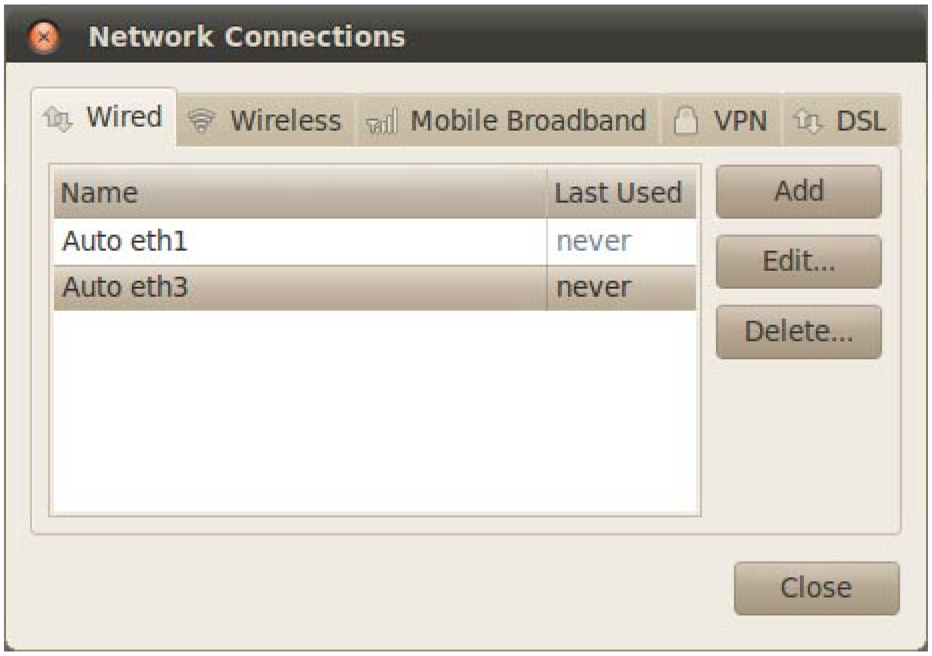
   a. Plug your Sourcery probe into your Linux host and ensure the TX/RX led is flashing green/off.

   b. Right-click on your desktop network icon to display the network manager GUI Application menu.

   c. Select **Edit Connections...**.

   d. In the Network Connections Dialog select the **Wired** tab, then select the interface associated with the probe and click **Edit...**. The highest numbered interface is most likely the probe, but you can confirm this in the Edit dialog.

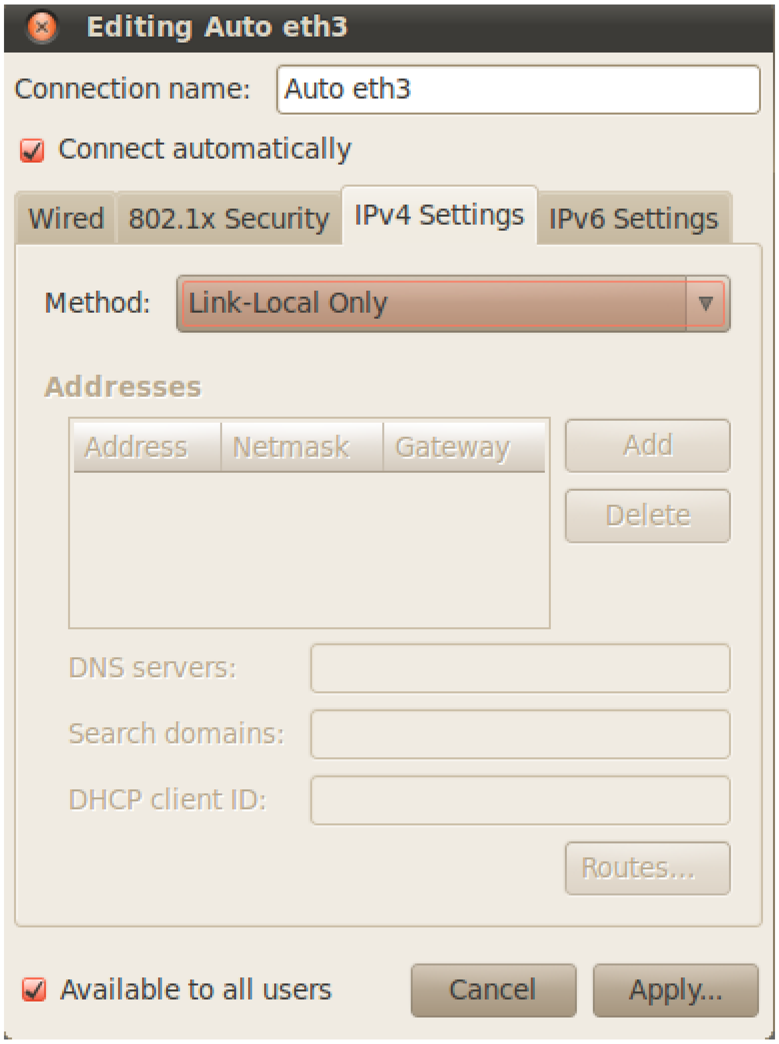
Figure A-1. Network Connections Dialog



Name	Last Used
Auto eth1	never
Auto eth3	never

   e. In the Editing Auto ethX dialog, verify the MAC address ends in 22:33:44. This ensures the interface you are editing is a virtual interface and is most likely the probe. If it is not, go back to the previous dialog and select a different Wired interface. Then, select the **IPv4 Settings** tab and change the method to **Link-Local Only**.

Figure A-2. Editing Network Connection Dialog



Editing Auto eth3

Connection name: Auto eth3

☑ Connect automatically

Wired | 802.1x Security | IPv4 Settings | IPv6 Settings

Method: Link-Local Only

Addresses

Address	Netmask	Gateway

Add

Delete

DNS servers:

Search domains:

DHCP client ID:

Routes...

☑ Available to all users

Cancel

Apply...

f. Click **Apply**.

3. Verify an IP address was assigned to the host interface:

- Open a terminal window and use the ifconfig command to verify the Local Link setup. Local link IP addresses will be 169.254.xxx.xxx with a netmask of 255.255.0.0. You can see that the address 169.254.5.96 was automatically allocated. The probe should now be discoverable.

```
ubuntu: $ ifconfig
eth1 Link encap:Ethernet HWaddr 00:0c:29:b1:18:e2
inet addr:192.168.98.150 Bcast:192.168.98.255 Mask:255.255.255.0
inet6 addr: fe80::20c:29ff:feb1:18e2/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:40767 errors:14 dropped:23 overruns:0 frame:0
TX packets:17759 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:49830008 (49.8 MB) TX bytes:1110165 (1.1 MB)
```

```
Interrupt:19 Base address:0x2000

eth3 Link encap:Ethernet HWaddr 00:07:e9:22:33:44
inet addr:169.254.5.96 Bcast:169.254.255.255 Mask:255.255.0.0
inet6 addr: fe80::207:e9ff:fe22:3344/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1472 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:95 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:21553 (21.5 KB)
```

Sourcery Probe Professional Settings

For instructions on configuring the communication settings for Sourcery Probe Professional, refer to the *Mentor Embedded Sourcery Probe Hardware Manual*.

If you need to perform a system-level firmware update, see the *Mentor Embedded Sourcery Probe Professional Hardware Manual*.

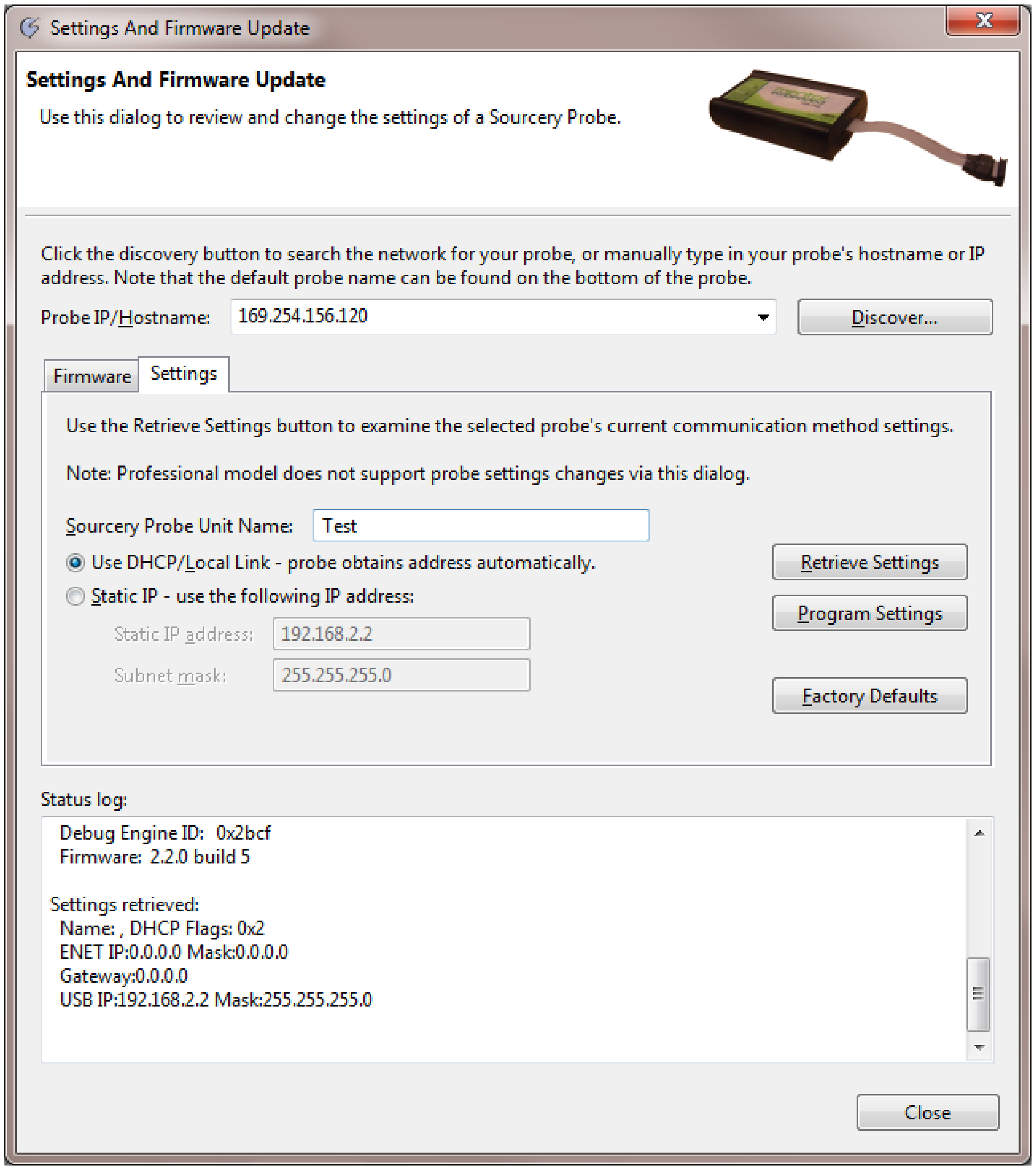
Settings and Firmware Update Dialog Box

To access: From Sourcery CodeBench, select **Run** > **Sourcery Probe** > **Settings and Firmware Update**.

Use this dialog box to:

- Update the firmware for all MESP Professional Models and ARM and MIPS MESP Personal probes.
- Configure network IP Settings for Mentor Embedded Sourcery Probe Personal.

Figure A-3. Settings and Firmware Update Dialog Box


Settings And Firmware Update
Settings And Firmware Update
Use this dialog to review and change the settings of a Sourcery Probe.
Click the discovery button to search the network for your probe, or manually type in your probe's hostname or IP address. Note that the default probe name can be found on the bottom of the probe.
Probe IP/Hostname: 169.254.156.120
Discover...
Firmware
Settings
Use the Retrieve Settings button to examine the selected probe's current communication method settings.
Note: Professional model does not support probe settings changes via this dialog.
Sourcery Probe Unit Name: Test
Use DHCP/Local Link - probe obtains address automatically.
Static IP - use the following IP address:
Static IP address: 192.168.2.2
Subnet mask: 255.255.255.0
Retrieve Settings
Program Settings
Factory Defaults
Status log:
Close

```
  Debug Engine ID:  0x2bcf
  Firmware:  2.2.0 build 5

Settings retrieved:
  Name: , DHCP Flags: 0x2
  ENET IP:0.0.0.0 Mask:0.0.0.0
  Gateway:0.0.0.0
  USB IP:192.168.2.2 Mask:255.255.255.0
```

Table A-2. Settings and Firmware Update Dialog Box Contents

Field	Description
Probe IP/Hostname	Enter the IP address or hostname for the Sourcery Probe.

Table A-2. Settings and Firmware Update Dialog Box Contents

Field	Description
Discover...	Click to view a list of all Sourcery Probe devices currently available on the local subnet. This opens the Sourcery Probe Discovery Dialog Box and scans all available network adapters for Sourcery Probes.
Settings Tab (See Table A-3)	Use the options on this tab to configure the ethernet settings for Sourcery Probe Personal.
Firmware Tab (See Figure A-4)	Use this tab to perform a firmware update that changes the software version running on both Sourcery Probe Personal and Professional.

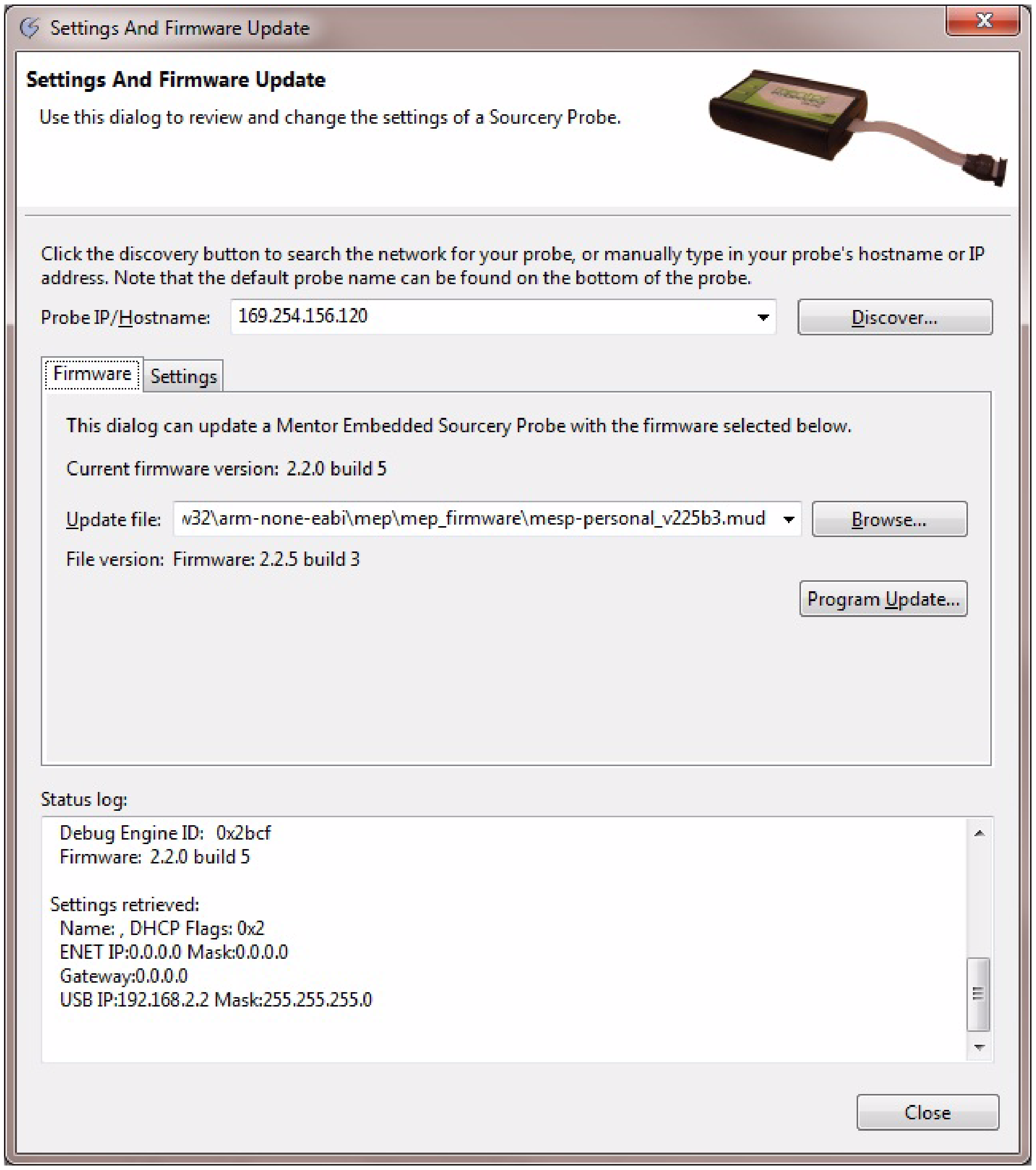
Table A-3. Mentor Embedded Sourcery Probe Update Dialog Box - Settings Tab

Field	Description
Sourcery Probe Name	Assign a name to the probe.
Use DHCP/Local Link	Select for the probe to obtain the IP address configuration automatically from the DHCP server on the network. This is the default configuration.

Table A-3. Mentor Embedded Sourcery Probe Update Dialog Box - Settings Tab

Field	**Description**
Static IP	Select to use the static IP address configuration. This requires you to enter the following values: • Static IP address, • Subnet mask These values should be assigned by your network administrator. CAUTION: Do not use this option unless you require it. The **Local Link** mode is recommended for the probe, even when the PC network adapter is set to static mode. New static IP settings can result in the probe becoming inaccessible if the settings are incorrect or incompatible with the network. If the probe is inaccessible after programming new settings, you must change your network adapter settings to match the IP range used by the probe.
Retrieve Settings	Displays the existing settings for the selected probe.
Program Settings	Modifies the settings on the probe and reboots it. **NOTE**: New static IP settings can result in the probe becoming inaccessible if the settings are incorrect or incompatible with the network. If the probe is inaccessible after programming new settings, you must change your network adapter settings to match the IP range used by the probe.
Factory Defaults	Restores the default settings on the probe.

Figure A-4. Settings and Firmware Update Dialog Box - Firmware Tab



Settings And Firmware Update

Use this dialog to review and change the settings of a Sourcery Probe.

Click the discovery button to search the network for your probe, or manually type in your probe's hostname or IP address. Note that the default probe name can be found on the bottom of the probe.

Probe IP/Hostname: 169.254.156.120

This dialog can update a Mentor Embedded Sourcery Probe with the firmware selected below.

Current firmware version: 2.2.0 build 5

Update file: v32\arm-none-eabi\mep\mep_firmware\mesp-personal_v225b3.mud

File version: Firmware: 2.2.5 build 3

Status log:

```
  Debug Engine ID:  0x2bcf
  Firmware:  2.2.0 build 5

Settings retrieved:
  Name: , DHCP Flags: 0x2
  ENET IP:0.0.0.0 Mask:0.0.0.0
  Gateway:0.0.0.0
  USB IP:192.168.2.2 Mask:255.255.255.0
```

Table A-4. Mentor Embedded Sourcery Probe Update Dialog Box - Firmware Tab

Field	Description
Current firmware version	This shows the version number of the firmware currently used by Sourcery Probe.
Update file	This is the path to the new firmware update file. This field also stores previously used update files, available in the dropdown list.

Table A-4. Mentor Embedded Sourcery Probe Update Dialog Box - Firmware Tab

Field	Description
Browse	Click **Browse** to select a different update file on the file system. By default this opens a dialog displaying the directory with the update files included with the Sourcery CodeBench installation. Update file naming conventions are: • `mesp-personal_vXYZbN` for Sourcery Probe Personal and Mentor Embedded USB-JTAG Probe • `mesp-professional_vXYZbN` for Sourcery Probe Professional and Mentor Embedded Giga-JTAG Probe
File version	This displays the update file version.
Program update	Updates the firmware on the selected probe. At the end of the update process the probe is rebooted automatically. It will use the new firmware after it reboots. **NOTE:** Do not interrupt or cancel the update process.

Related Topics

Updating the Sourcery Probe Firmware Using CodeBench

Sourcery Probe Personal Settings (ARM and MIPS Only)

Sourcery Probe Professional Settings

Sourcery Probe Discovery Dialog Box

Use this dialog box to display a list of Sourcery Probes available on the network. It works by sending a unique multicast packet to the network and listening for replies from the Sourcery Probes.

To access: From the Settings and Firmware Update Dialog Box— Settings Tab, click **Discover**.

Note

If the probe you want to use is not shown in the list, click the **Rescan** button (Rescan). If it still is not found, then you must manually enter the IP address or hostname of your probe. Some common reasons why a probe may not be discoverable include:

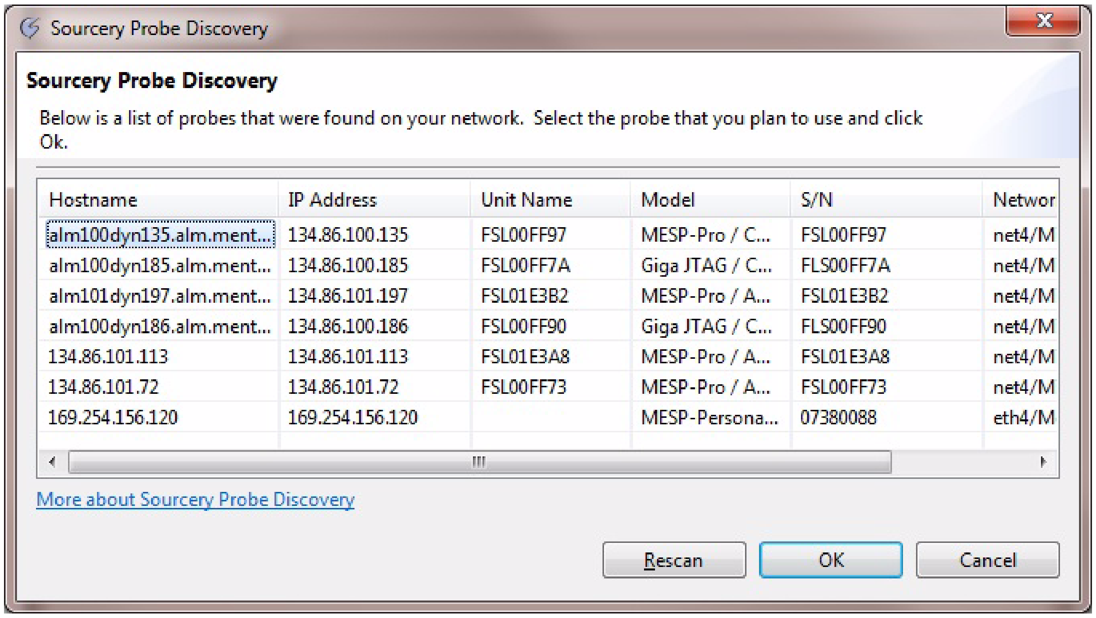
- If your probe is not on your current host machine's subnet, or your network adapter is not configured for multi-cast, it might not be discoverable.

- Your probe also might not be discoverable if the routers servicing your network are blocking multi-cast packets, a very common practice among companies.

- Many VPN environments also block multi-cast packets.

- If multi-casting is blocked, you must manually enter the IP address or hostname of your probe.

Figure A-5. Sourcery Probe Discovery Dialog Box



Sourcery Probe Discovery

Below is a list of probes that were found on your network. Select the probe that you plan to use and click Ok.

Hostname	IP Address	Unit Name	Model	S/N	Networ
alm100dyn135.alm.ment...	134.86.100.135	FSL00FF97	MESP-Pro / C...	FSL00FF97	net4/M
alm100dyn185.alm.ment...	134.86.100.185	FSL00FF7A	Giga JTAG / C...	FLS00FF7A	net4/M
alm101dyn197.alm.ment...	134.86.101.197	FSL01E3B2	MESP-Pro / A...	FSL01E3B2	net4/M
alm100dyn186.alm.ment...	134.86.100.186	FSL00FF90	Giga JTAG / C...	FLS00FF90	net4/M
134.86.101.113	134.86.101.113	FSL01E3A8	MESP-Pro / A...	FSL01E3A8	net4/M
134.86.101.72	134.86.101.72	FSL00FF73	MESP-Pro / A...	FSL00FF73	net4/M
169.254.156.120	169.254.156.120		MESP-Persona...	07380088	eth4/M

Table A-5. Sourcery Probe Discovery Contents

Field	Description
Rescan	Because multicast packets are subject to collisions on the network, click **Rescan** if you don't see your probe in the list.

Table A-5. Sourcery Probe Discovery Contents

Field	Description
OK	Select a probe from the list and click **OK** to return to the Settings and Firmware Update Dialog Box.
Cancel	Cancel the discovery option and close the dialog box without selecting any probe.

Related Topics

Settings and Firmware Update Dialog Box

Sourcery Probe Serial Port Console

To access the serial port for the probe connect it to the USB port of the host and use the Terminal view in Sourcery CodeBench (select **Window** > **Show View** > **Other** - **Terminal**) to connect to the serial port. Choose **Serial** as the connection type, then set the serial port settings as follows:

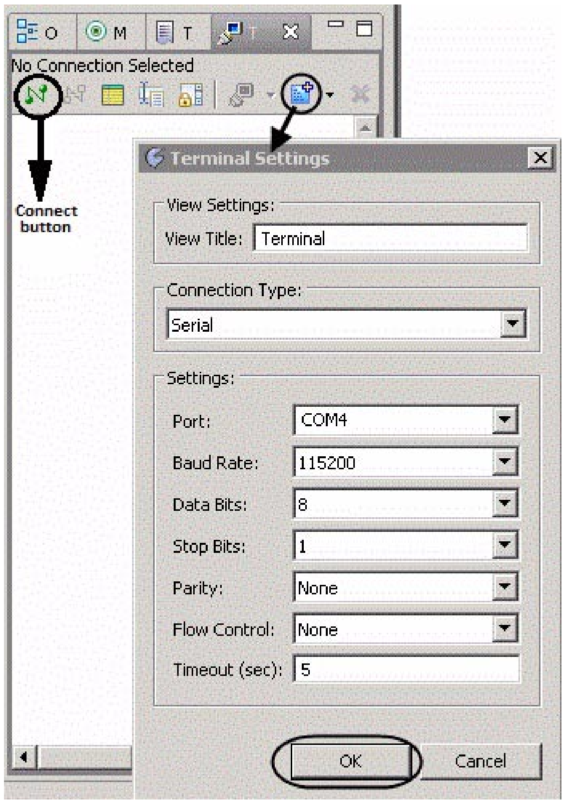
- Baud: 115200
- Data Bits: 8
- Stop Bits: 1
- Parity: None
- Flow control: none

After applying these settings as shown in Figure A-6, click the **Connect** button to open the connection, and then press the **<ESC>** key for the probe's console menu.

Note

The Sourcery Probe Personal PowerPC models do not support a virtual serial console.

Figure A-6. New Terminal Connection using Sourcery CodeBench Terminal View



View Settings:
View Title: Terminal

Connection Type:
Serial

Settings:
Port: COM4
Baud Rate: 115200
Data Bits: 8
Stop Bits: 1
Parity: None
Flow Control: None
Timeout (sec): 5

For more information on the USB serial port, refer to the *Mentor Embedded Sourcery Probe Personal Hardware Manual* and the *Mentor Embedded Sourcery Probe Professional Hardware Manual*.

Third-Party Information

This section provides information on open source and third-party software that may be included in the Mentor Embedded Sourcery Probe product.

- This software application may include dlfcn-win32 version r19 third-party software. dlfcn-win32 version r19 is distributed under the terms of the GNU Lesser General Public License version 2.1 and is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the license for the specific language governing rights and limitations under the license. You can view a copy of the license at:

  <install_dir>/share/doc/sourceryg++-arm-none-eabi/pdf/legal/gnu_lgpl_2.1.pdf.

  To obtain a copy of the dlfcn-win32 version r19 source code, send a request to request_sourcecode@mentor.com. This offer shall only be available for three years from the date Mentor Graphics Corporation first distributed dlfcn-win32 version r19.

- This software application may include pthreads-win32 version 2.9.1 third-party software. pthreads-win32 version 2.9.1 is distributed under the terms of the GNU Lesser General Public License version 2.1 and is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the license for the specific language governing rights and limitations under the license. You can view a copy of the license at:

  <install_dir>/share/doc/sourceryg++-arm-none-eabi/pdf/legal/gnu_lgpl_2.1.pdf.

  To obtain a copy of the pthreads-win32 version 2.9.1 source code, send a request to request_sourcecode@mentor.com. This offer shall only be available for three years from the date Mentor Graphics Corporation first distributed pthreads-win32 version 2.9.1.

Embedded Software and Hardware License Agreement

The latest version of the Embedded Software and Hardware License Agreement is available on-line at:
www.mentor.com/eshla

IMPORTANT INFORMATION

USE OF ALL PRODUCTS IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF PRODUCTS INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

EMBEDDED SOFTWARE AND HARDWARE LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Products (as defined in Section 1) between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Products received electronically, certify destruction of Products and all accompanying items within five days after receipt of such Products and receive a full refund of any license fee paid.

1. **Definitions.** As used in this Agreement and any applicable quotation, supplement, attachment and/or addendum ("Addenda"), these terms shall have the following meanings:

1.1. "Customer's Product" means Customer's end-user product identified by a unique SKU (including any Related SKUs) in an applicable Addenda that is developed, manufactured, branded and shipped solely by Customer or an authorized manufacturer or subcontractor on behalf of Customer to end-users or consumers;

1.2. "Developer" means a unique user, as identified by a unique user identification number, with access to Embedded Software at an authorized Development Location. A unique user is an individual who works directly with the embedded software in source code form, or creates, modifies or compiles software that ultimately links to the Embedded Software in Object Code form and is embedded into Customer's Product at the point of manufacture;

1.3. "Development Location" means the location where Products may be used as authorized in the applicable Addenda;

1.4. "Development Tools" means the software that may be used by Customer for building, editing, compiling, debugging or prototyping Customer's Product;

1.5. "Embedded Software" means Software that is embeddable;

1.6. "End-User" means Customer's customer;

1.7. "Executable Code" means a compiled program translated into a machine-readable format that can be loaded into memory and run by a certain processor;

1.8. "Hardware" means a physically tangible electro-mechanical system or sub-system and associated documentation;

1.9. "Linkable Object Code" or "Object Code" means linkable code resulting from the translation, processing, or compiling of Source Code by a computer into machine-readable format;

1.10. "Mentor Embedded Linux" or "MEL" means Mentor Graphics' tools, source code, and recipes for building Linux systems;

1.11. "Open Source Software" or "OSS" means software subject to an open source license which requires as a condition for redistribution of such software, including modifications thereto, that the: (i) redistribution be in source code form or be made available in source code form; (ii) redistributed software be licensed to allow the making of derivative works; or (iii) redistribution be at no charge;

1.12. "Processor" means the specific microprocessor to be used with Software and implemented in Customer's Product;

1.13. "Products" means Software, Term-Licensed Products and/or Hardware;

1.14. "Proprietary Components" means the components of the Products that are owned and/or licensed by Mentor Graphics and are not subject to an Open Source Software license, as more fully set forth in the product documentation provided with the Products;

1.15. “Redistributable Components” means those components that are intended to be incorporated or linked into Customer’s Linkable Object Code developed with the Software, as more fully set forth in the documentation provided with the Products;

1.16. “Related SKU” means two or more Customer Products identified by logically-related SKUs, where there is no difference or change in the electrical hardware or software content between such Customer Products;

1.17. “Software” means software programs, Embedded Software and/or Development Tools, including any updates, modifications, revisions, copies, documentation and design data that are licensed under this Agreement;

1.18. “Source Code” means software in a form in which the program logic is readily understandable by a human being;

1.19. “Sourcery CodeBench Software” means Mentor Graphics’ Development Tool for C/C++ embedded application development;

1.20. “Sourcery VSIPL++” is Software providing C++ classes and functions for writing embedded signal processing applications designed to run on one or more processors;

1.21. “Stock Keeping Unit” or “SKU” is a unique number or code used to identify each distinct product, item or service available for purchase;

1.22. “Subsidiary” means any corporation more than 50% owned by Customer, excluding Mentor Graphics competitors. Customer agrees to fulfill the obligations of such Subsidiary in the event of default. To the extent Mentor Graphics authorizes any Subsidiary’s use of Products under this Agreement, Customer agrees to ensure such Subsidiary’s compliance with the terms of this Agreement and will be liable for any breach by a Subsidiary; and

1.23. “Term-Licensed Products” means Products licensed to Customer for a limited time period (“Term”).

2. **Orders, Fees and Payment.**

2.1. To the extent Customer (or if agreed by Mentor Graphics, Customer’s appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement (“Order(s)”), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement and any applicable Addenda, whether or not these documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order will not be effective unless agreed in writing by an authorized representative of Customer and Mentor Graphics.

2.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. All invoices will be sent electronically to Customer on the date stated on the invoice unless otherwise specified in an Addendum. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice(s). Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer’s sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer’s behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.

2.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics’ delivery of Software by electronic means is subject to Customer’s provision of both a primary and an alternate e-mail address.

3. **Grant of License.**

3.1. The Products installed, downloaded, or otherwise acquired by Customer under this Agreement constitute or contain copyrighted, trade secret, proprietary and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software as described in the applicable Addenda. The limited licenses granted under the applicable Addenda shall continue until the expiration date of Term-Licensed Products or termination in accordance with Section 12 below, whichever occurs first. **Mentor Graphics does NOT grant Customer any right to (a) sublicense or (b) use Software beyond the scope of this Section without first signing a separate agreement or Addenda with Mentor Graphics for such purpose.**

3.2. License Type. The license type shall be identified in the applicable Addenda.

3.2.1. Development License: During the Term, if any, Customer may modify, compile, assemble and convert the applicable Embedded Software Source Code into Linkable Object Code and/or Executable Code form by the number of Developers specified, for the Processor(s), Customer’s Product(s) and at the Development Location(s) identified in the applicable Addenda.

3.2.2. End-User Product License: During the Term, if any, and unless otherwise specified in the applicable Addenda, Customer may incorporate or embed an Executable Code version of the Embedded Software into the specified number of copies of Customer's Product(s), using the Processor Unit(s), and at the Development Location(s) identified in the applicable Addenda. Customer may manufacture, brand and distribute such Customer's Product(s) worldwide to its End-Users.

3.2.3. Internal Tool License: During the Term, if any, Customer may use the Development Tools solely: (a) for internal business purposes and (b) on the specified number of computer work stations and sites. Development Tools are licensed on a per-seat or floating basis, as specified in the applicable Addenda, and shall not be distributed to others or delivered in Customer's Product(s) unless specifically authorized in an applicable Addenda.

3.2.4. Sourcery CodeBench Professional Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software (i) if the license is a node-locked license, by a single user who uses the Software on up to two machines provided that only one copy of the Software is in use at any one time, or (ii) if the license is a floating license, by the authorized number of concurrent users on one or more machines provided that only the authorized number of copies of the Software are in use at any one time, and (b) distribute the Redistributable Components of the Software in Executable Code form only and only as part of Customer's Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.

3.2.5. Sourcery CodeBench Standard Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software by a single user who uses the Software on up to two machines provided that only one copy of the Software is in use at any one time, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer's Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.

3.2.6. Sourcery CodeBench Personal Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software by a single user who uses the Software on one machine, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.

3.2.7. Sourcery CodeBench Academic Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software for non-commercial, academic purposes only by a single user who uses the Software on one machine, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.

3.3. Mentor Graphics may from time to time, in its sole discretion, lend Products to Customer. For each loan, Mentor Graphics will identify in writing the quantity and description of Software loaned, the authorized location and the Term of the loan. Mentor Graphics will grant to Customer a temporary license to use the loaned Software solely for Customer's internal evaluation in a non-production environment. Customer shall return to Mentor Graphics or delete and destroy loaned Software on or before the expiration of the loan Term. Customer will sign a certification of such deletion or destruction if requested by Mentor Graphics.

4. **Beta Code.**

4.1. Portions or all of certain Products may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and Customer's use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form.

4.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.

4.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 4.3 shall survive termination of this Agreement.

5. **Restrictions on Use.**

5.1. Customer may copy Software only as reasonably necessary to support the authorized use, including archival and backup purposes. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Except where embedded in Executable Code form in Customer's Product, Customer shall maintain a record of the number and location of all copies of Software, including copies merged with other software and products, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer's employees, authorized manufacturers or authorized contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics immediate written notice of any unauthorized disclosure or use of the Products as soon as Customer learns or becomes aware of such unauthorized disclosure or use.

5.2. Customer acknowledges that the Products provided hereunder may contain Source Code which is proprietary and its confidentiality is of the highest importance and value to Mentor Graphics. Customer acknowledges that Mentor Graphics may be seriously harmed if such Source Code is disclosed in violation of this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive any Source Code from Products that are not provided in Source Code form. Except as embedded in Executable Code in Customer's Product and distributed in the ordinary course of business, in no event shall Customer provide Products to Mentor Graphics competitors. Log files, data files, rule files and script files generated by or for the Software (collectively "Files") constitute and/or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Under no circumstances shall Customer use Products or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, or disclose to any third party the results of, or information pertaining to, any benchmark.

5.3. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent, which shall not be unreasonably withheld, and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.

5.4. Notwithstanding any provision in an OSS license agreement applicable to a component of the Sourcery CodeBench Software that permits the redistribution of such component to a third party in Source Code or binary form, Customer may not use any Mentor Graphics trademark, whether registered or unregistered, in connection with such distribution, and may not recompile the Open Source Software components with the --with-pkgversion or --with-bugurl configuration options that embed Mentor Graphics' trademarks in the resulting binary.

5.5. The provisions of this Section 5 shall survive the termination of this Agreement.

6. **Support Services.**

6.1. Except as described in Sections 6.2, 6.3 and 6.4 below, and unless otherwise specified in any applicable Addenda to this Agreement, to the extent Customer purchases support services, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased in accordance with Mentor Graphics' then-current End-User Software Support Terms located at http://supportnet.mentor.com/about/legal/.

6.2. To the extent Customer purchases support services for Sourcery CodeBench Software, support will be provided solely in accordance with the provisions of this Section 6.2. Mentor Graphics shall provide updates and technical support to Customer as described herein only on the condition that Customer uses the Executable Code form of the Sourcery CodeBench Software for internal use only and/or distributes the Redistributable Components in Executable Code form only (except as provided in a separate redistribution agreement with Mentor Graphics or as required by the applicable Open Source license). Any other distribution by Customer of the Sourcery CodeBench Software (or any component thereof) in any form, including distribution permitted by the applicable Open Source license, shall automatically terminate any remaining support term. Subject to the foregoing and the payment of support fees, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased in accordance with Mentor Graphics' then-current Sourcery CodeBench Software Support Terms located at http://www.mentor.com/codebench-support-legal.

6.3. To the extent Customer purchases support services for Sourcery VSIPL++, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased solely in accordance with Mentor Graphics' then-current Sourcery VSIPL++ Support Terms located at http://www.mentor.com/vsipl-support-legal.

6.4. To the extent Customer purchases support services for Mentor Embedded Linux, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased solely in accordance with Mentor Graphics' then-current Mentor Embedded Linux Support Terms located at http://www.mentor.com/mel-support-legal.

7. **Third Party and Open Source Software.** Products may contain Open Source Software or code distributed under a proprietary third party license agreement. Please see applicable Products documentation, including but not limited to license notice files, header files or source code for further details. Please see the applicable Open Source Software license(s) for additional rights and obligations governing your use and distribution of Open Source Software. Customer agrees that it shall not subject any Product provided by Mentor Graphics under this Agreement to any Open Source Software license that does not otherwise apply to such Product. In the event of conflict between the terms of this Agreement, any Addenda and an applicable OSS or proprietary third party agreement, the OSS or proprietary third party agreement will control solely with respect to the OSS or proprietary third party software component. The provisions of this Section 7 shall survive the termination of this Agreement.

8. **Limited Warranty.**

   8.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual and/or specification. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Products under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification or improper installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY, PROVIDED CUSTOMER HAS OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; OR (B) PRODUCTS PROVIDED AT NO CHARGE, WHICH ARE PROVIDED "AS IS" UNLESS OTHERWISE AGREED IN WRITING.

   8.2. THE WARRANTIES SET FORTH IN THIS SECTION 8 ARE EXCLUSIVE TO CUSTOMER AND DO NOT APPLY TO ANY END-USER. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, WITH RESPECT TO PRODUCTS OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

9. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, AND EXCEPT FOR EITHER PARTY'S BREACH OF ITS CONFIDENTIALITY OBLIGATIONS, CUSTOMER'S BREACH OF LICENSING TERMS OR CUSTOMER'S OBLIGATIONS UNDER SECTION 10, IN NO EVENT SHALL: (A) EITHER PARTY OR ITS RESPECTIVE LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF SUCH PARTY OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES; AND (B) EITHER PARTY OR ITS RESPECTIVE LICENSORS' LIABILITY UNDER THIS AGREEMENT, INCLUDING, FOR THE AVOIDANCE OF DOUBT, LIABILITY FOR ATTORNEYS' FEES OR COSTS, EXCEED THE GREATER OF THE FEES PAID OR OWING TO MENTOR GRAPHICS FOR THE PRODUCT OR SERVICE GIVING RISE TO THE CLAIM OR $500,000 (FIVE HUNDRED THOUSAND U.S. DOLLARS). NOTWITHSTANDING THE FOREGOING, IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 9 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

10. **Hazardous Applications.**

    10.1. Customer agrees that Mentor Graphics has no control over Customer's testing or the specific applications and use that Customer will make of Products. Mentor Graphics Products are not specifically designed for use in the operation of nuclear facilities, aircraft navigation or communications systems, air traffic control, life support systems, medical devices or other applications in which the failure of Mentor Graphics Products could lead to death, personal injury, or severe physical or environmental damage ("Hazardous Applications").

    10.2. CUSTOMER ACKNOWLEDGES IT IS SOLELY RESPONSIBLE FOR TESTING PRODUCTS USED IN HAZARDOUS APPLICATIONS AND SHALL BE SOLELY LIABLE FOR ANY DAMAGES RESULTING FROM SUCH USE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF PRODUCTS IN ANY HAZARDOUS APPLICATIONS.

    10.3. CUSTOMER AGREES TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE OR LIABILITY, INCLUDING REASONABLE ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH THE USE OF PRODUCTS AS DESCRIBED IN SECTION 10.1.

    10.4. THE PROVISIONS OF THIS SECTION 10 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

11. **Infringement.**

11.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay any costs and damages finally awarded against Customer that are attributable to the action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

11.2. If a claim is made under Subsection 11.1 Mentor Graphics may, at its option and expense, and in addition to its obligations under Section 11.1, either (a) replace or modify the Product so that it becomes noninfringing; or (b) procure for Customer the right to continue using the Product. If Mentor Graphics determines that neither of those alternatives is financially practical or otherwise reasonably available, Mentor Graphics may require the return of the Product and refund to Customer any purchase price or license fee(s) paid.

11.3. Mentor Graphics has no liability to Customer if the claim is based upon: (a) the combination of the Product with any product not furnished by Mentor Graphics, where the Product itself is not infringing; (b) the modification of the Product other than by Mentor Graphics or as directed by Mentor Graphics, where the unmodified Product would not infringe; (c) the use of the infringing Product when Mentor Graphics has provided Customer with a current unaltered release of a non-infringing Product of substantially similar functionality in accordance with Subsection 11.2(a); (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells, where the Product itself is not infringing; (f) any Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; (h) Open Source Software, except to the extent that the infringement is directly caused by Mentor Graphics' modifications to such Open Source Software; or (i) infringement by Customer that is deemed willful. In the case of (i), Customer shall reimburse Mentor Graphics for its reasonable attorneys' fees and other costs related to the action.

11.4. THIS SECTION 11 IS SUBJECT TO SECTION 9 ABOVE AND STATES: (A) THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND (B) CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.

12. **Termination and Effect of Termination.** If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized Term.

12.1. Termination for Breach. This Agreement shall remain in effect until terminated in accordance with its terms. Mentor Graphics may terminate this Agreement and/or any licenses granted under this Agreement, and Customer will immediately discontinue use and distribution of Products, if Customer (a) commits any material breach of any provision of this Agreement and fails to cure such breach upon 30-days prior written notice; or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination. For the avoidance of doubt, nothing in this Section 12 shall be construed to prevent Mentor Graphics from seeking immediate injunctive relief in the event of any threatened or actual breach of Customer's obligations hereunder.

12.2. Effect of Termination. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination or expiration of the Term, Customer will discontinue use and/or distribution of Products, and shall return Hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form, except to the extent an Open Source Software license conflicts with this Section 12.2 and permits Customer's continued use of any Open Source Software portion or component of a Product. Upon termination for Customer's breach, an End-User may continue its use and/or distribution of Customer's Product so long as: (a) the End-User was licensed according to the terms of this Agreement, if applicable to such End-User, and (b) such End-User is not in breach of its agreement, if applicable, nor a party to Customer's breach.

13. **Export.** The Products provided hereunder are subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export Products in any manner without first obtaining all necessary approval from appropriate local and United States government agencies. Customer acknowledges that the regulation of product export is in continuous modification by local governments and/or the United States Congress and administrative agencies. Customer agrees to complete all documents and to meet all requirements arising out of such modifications.

14. **U.S. Government License Rights.** Software was developed entirely at private expense. All Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to US FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in this Agreement, except for provisions which are contrary to applicable mandatory federal laws.

15. **Third Party Beneficiary.** For any Products licensed under this Agreement and provided by Customer to End-Users, Mentor Graphics or the applicable licensor is a third party beneficiary of the agreement between Customer and End-User. Mentor

Graphics Corporation, Mentor Graphics (Ireland) Limited, and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.

16. **Review of License Usage.** Customer will monitor the access to and use of Software. With prior written notice, during Customer's normal business hours, and no more frequently than once per calendar year, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system, records, accounts and sublicensing documents deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all Customer information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. Such license review shall be at Mentor Graphics' expense unless it reveals a material underpayment of fees of five percent or more, in which case Customer shall reimburse Mentor Graphics for the costs of such license review. Customer shall promptly pay any such fees. If the license review reveals that Customer has made an overpayment, Mentor Graphics has the option to either provide the Customer with a refund or credit the amount overpaid to Customer's next payment. The provisions of this Section 16 shall survive the termination of this Agreement.

17. **Controlling Law, Jurisdiction and Dispute Resolution.** This Agreement shall be governed by and construed under the laws of the State of California, USA, excluding choice of law rules. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the state and federal courts of California, USA. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer or its Subsidiary in the jurisdiction where Customer's or its Subsidiary's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.

18. **Severability.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

19. **Miscellaneous.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements, including but not limited to any purchase order terms and conditions. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.

Rev. 120305, Part No. 252061